Please comment on slides with anything unclear, incorrect or suggestions for improvement
https://drive.google.com/file/d/1DD5tgyaS8Nk_Wze97_IJpVBRQhXuwgT8/view?usp=drive_link

PV204 Security technologies

JavaCard optimizations, Secure Multiparty Computation and Trhreshold signatures

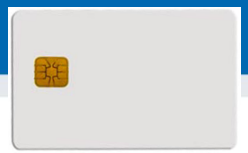
Petr Švenda ***svenda@fi.muni.cz*** ***@rngsec***
Centre for Research on Cryptography and Security, Masaryk University

(part of MPC slides done by Antonín Dufka)

BEST PRACTICES (FOR APPLET DEVELOPERS)

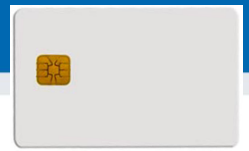
Quiz

1. Expect that your device is leaking in time/power channel. Which option will you use?
   - AES from hw coprocessor or software re-implementation?
   - Short-term sensitive data stored in EEPROM or RAM?
   - Persistent sensitive data in EEPROM or encrypted object?
   - Conditional jumps on sensitive value?
2. Expect that attacker can successfully induct faults (random change of bit(s) in device memory).
   - Suggest defensive options for applet's source code
   - Change in RAM, EEPROM, instruction pointer, CPU flags…



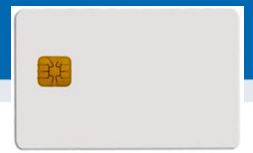
Security hints (1)

- Use algorithms/modes from JC API rather than your own implementation
  - JC API algorithms fast and protected in cryptographic hardware
  - general-purpose processor leaks more information (side-channels)
- Store session data in RAM
  - faster and more secure against power analysis
  - EEPROM has limited number of rewrites (10^5 – 10^6 writes)
- Never store keys, PINs or sensitive data in primitive arrays
  - use specialized objects like OwnerPIN and Key
  - better protected against power, fault and memory read-out attacks
  - If not possible, generate random key in Key object, encrypt large data with this key and store only encrypted data
- Make checksum on stored sensitive data (=> detect faults)
  - check during applet selection (do not continue if data are corrupted)
  - possibly check also before sensitive operation with the data (but performance penalty)



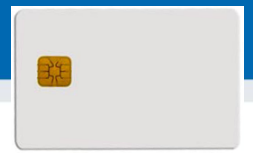
Security hints (2)

- Erase unused keys and sensitive arrays
  - use specialized method if exists (Key.clearKey())
  - or overwrite with random data (Random.generate())
  - Perform always before and after start of new session (new select, new device…)
- Use transactions to ensure atomic operations
  - power supply can be interrupted inside code execution
  - be aware of attacks by interrupted transactions - rollback attack
- Do not use conditional jumps with sensitive data
  - branching after condition is recognizable with power analysis => timing/power leakage



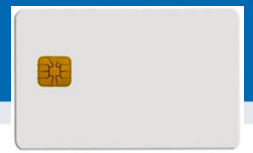
Security hints (3)

- Allocate all necessary resources in constructor
  - applet installation usually in trusted environment
  - prevents attacks based on limited available resources later during applet use
- Don’t use static attributes (except constants)
  - Static attribute is shared between multiple instances of applet (bypasses applet firewall)
  - Static pointer to array/engine filled by dynamic allocation cannot be removed until package is removed from card (memory “leak”)
- Use automata-based programming model
  - well defined states (e.g., user PIN verified)
  - well defined transitions and allowed method calls



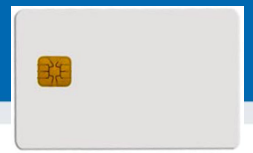
Security hints (4)

- Treat exceptions properly
  - Do not let uncaught native exceptions to propagate away from the card
    - 0x6f00 emitted – unclear what caused it from the terminal side
    - Your applet is unaware of the exception (fault induction attack in progress?)
  - Do not let your code to cause basic exceptions like OutOfBoundsException or NullPointerExceptions…
    - Slow handling of exceptions in general
    - Code shall not depend on triggering lower-layer defense (like memory protection)



Security hints: fault induction (1)

- Cryptographic algorithms are sensitive to fault induction
  - Single signature with fault from RSA-CRT may leak the private key
  - Perform operation twice and compare results
  - Perform reverse operation and compare (e.g., verify after sign)
- Use constants with large hamming distance
  - Induced fault in variable will likely cause unknown value
  - Use 0xA5 and 0x5A instead of 0 and 1 (correspondingly for more)
  - Don’t use values 0x00 and 0xff (easier to force all bits to 0 or 1)
- Check that all sub-functions were executed [Fault.Flow]
  - Fault may force program stack or stack to skip some code
  - Idea: Add defined value to flow counter inside target sub-function, check later for expected sum. Add also in branches.



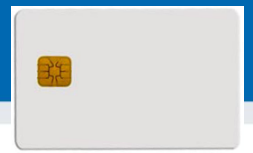
Security hints: fault induction (2)

- Replace single condition check by complementary check
  - **`conditionalValue`** is sensitive value
  - Do not use boolean values for sensitive decisions

```
if (conditionalValue == 0x3CA5965A) { // enter critical path
  // . . .
  if (~conditionalValue != 0xC35A69A5) {
    faultDetect(); // fail if complement not equal to 0xC35A69A5
  }
  // . . .
}
```

- Verify number of actually performed loop iterations

```
int i;
for ( i = 0; i < n; i++ ) { // important loop that must be completed
//. . .
}
if (i != n) { // loop not completed
  faultDetect();
}
```

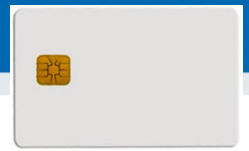


Security hints: fault induction (3)

- Insert random delays around sensitive operations
  - Randomization makes targeted faults more difficult
  - for loop with random number of iterations (for every run)
- Monitor and respond to detected induced faults
  - If fault is detected (using previous methods), increase fault counter.
  - Erase keys / lock card after reaching some threshold (~10)
    - Natural causes may occasionally cause fault => > 1

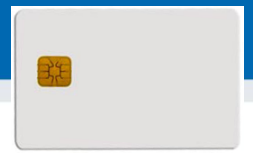
How and when to apply protections

- ✓ Does the device need protection?
- ✓ Understand the resistance of the hardware
- ✓ Identify potential weakness in design
- ✓ Select patterns to use
- ✓ Understand your compiler
- ✓ Code it
- ✓ Test the resistance of the device



Execution speed hints (1)

- Big difference between RAM and EEPROM memory
  - new allocates in EEPROM (persistent, but slow)
    - do not use EEPROM for temporary data
    - do not use for sensitive data (keys)
  - JCSystem.getTransientByteArray() for RAM buffer
  - local variables automatically in RAM
- Use algorithms from JavaCard API and utility methods
  - much faster, cryptographic co-processor
- Allocate all necessary resources in constructor
  - executed during installation (only once)
  - either you get everything you want or not install at all



Execution speed hints (2)

- Garbage collection limited or not available
  - do not use **`new`** except in constructor
- Use copy-free style of methods
  - foo(byte[] buffer, short start_offset, short length)
- Do not use recursion or frequent function calls
  - slow, function context overhead
- Do not use OO design extensively (slow)
- Keep Cipher or Signature objects initialized
  - if possible (e.g., fixed master key for subsequent derivation)
  - initialization with key takes non-trivial time

JCPROFILERNEXT – PERFORMANCE PROFILING, NON-CONSTANT TIME DETECTION

JCProfilerNext: on-card performance profiler

- Open-source on-card performance profiler (L. Zaoral)
  - https://github.com/lzaoral/JCProfilerNext
- Automatically instrumentation of provided JavaCard code
  - Conditional exception emitted on defined line of code
  - Spoon tool used https://spoon.gforge.inria.fr/
- Measures time to reach specific line (measured on client-side)
- Fully automatic, no need for special setup (only JavaCard + reader)
- Goals:
  - Help developer to identify parts for performance optimizations
  - Help to detect (significant) timing leakages
  - Insert “triggers” visible on side-channel analysis
  - Insert conditional breakpoints…

Instrumented code (Spoon)

```
// if m_perfStop equals to stopCondition, exception is thrown (trap hit)
public static void check(short stopCondition) {
    if (PM.m_perfStop == stopCondition) {
        ISOException.throwIt(stopCondition);
    }
}
```

```
private void example(APDU apdu) {
  PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_1);
  short count = Util.getShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA);
  PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_2);
  for (short i = 0; i < count; i++) {
    PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_3);
    short tmp = 0;
    PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_4);
    for (short k = 0; k < 50; k++) {
      PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_5);
      tmp++;
      PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_6);
    }
    PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_7);
  }
  PM.check(PMC.TRAP_example_Example_example_argb_javacard_framework_APDU_arge_8);
}
```

JCProfilerNext – timing profile of target line of code

example.Example.example2(javacard.framework.APDU)

TRAP_example_Example_example2_argb_javacard_framework_APDU_arge_12

Card ATR: 3BFA1800008131FE454A434F5033315632333298
Number of rounds: 1000
APDU header: 80010000
Input regex: 00[0-9A-F]{2}
Elapsed time: 0 days 00:00:02.814
Source measurements: measurements.csv

☐ Show explicit traps

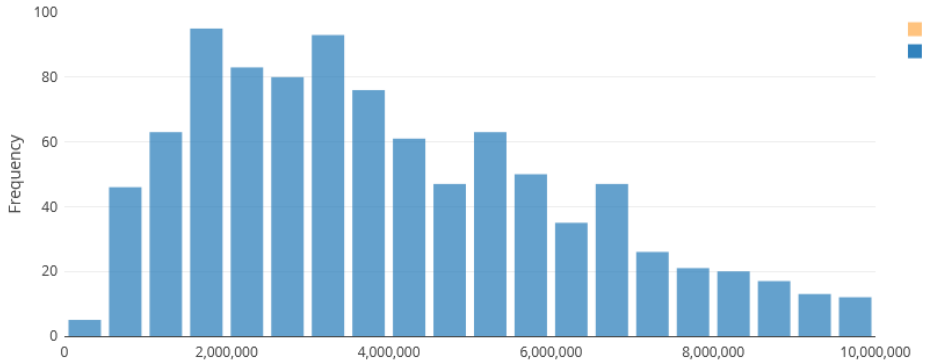
Avg µs

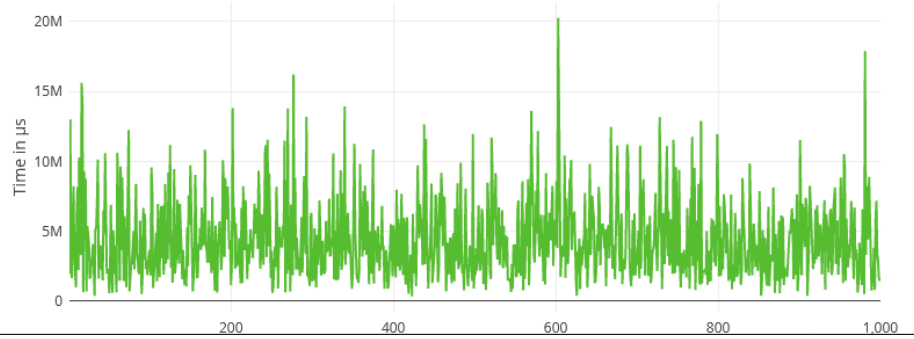
```
1  private void example2(APDU apdu) {
2      byte[] apdubuf = apdu.getBuffer();
3      short dataLen = apdu.setIncomingAndReceive();
4      // SET KEY VALUE
5      m_aesKey.setKey(apdubuf, ISO7816.OFFSET_CDATA);
6      // INIT CIPHERS WITH NEW KEY
7      m_encryptCipher.init(m_aesKey, Cipher.MODE_ENCRYPT);
8      m_decryptCipher.init(m_aesKey, Cipher.MODE_DECRYPT);
9      m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, ((short) (0x10)), m_ramArray, ((short) (0)));
10     m_decryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, ((short) (0x10)), m_ramArray, ((short) (0)));
11     // Hash input
12     m_hash.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray, ((short) (0)));
13     // GENERATE DATA
14     m_secureRandom.generateData(apdubuf, ISO7816.OFFSET_CDATA, ((short) (0x10)));
15     // Sign data
16     short signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA, ((byte) (dataLen)), m_ramArray, ((byte) (0)));
17     // Generate fresh key pair on-card
18     m_keyPair.genKeyPair();
19     m_publicKey = m_keyPair.getPublic();
20     m_privateKey = m_keyPair.getPrivate();
21     // INIT WITH PRIVATE KEY
22     m_sign.init(m_privateKey, Signature.MODE_SIGN);
23 }
```

Colour explanation

Currently selected trap | Trap was never reached | Trap was reached only sometimes

Click on a bin to get a list of corresponding inputs.





JCProfilerNext – memory consumption

opencrypto.jcmathlib.OCUnitTests()

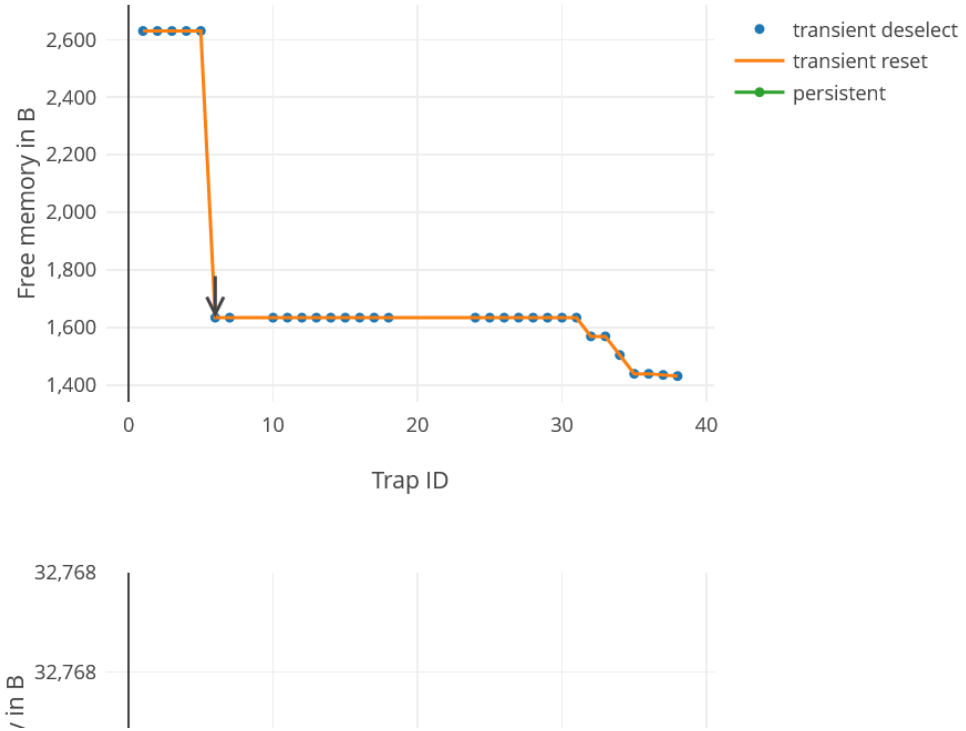
TRAP_opencrypto_jcmathlib_OCUnitTests_argb_arge_6

Mode: memory
Card ATR: 3B80800101
APDU header: measured during installation
Input: measured during installation
Elapsed time: 0 days 00:00:00.294
Source measurements: measurements.csv

☐ Show explicit traps

Diff in B

```
1  public OCUnitTests() {
2      OperationSupport.getInstance().setCard(OperationSupport.SIMULATOR);// TC
3      m_memoryInfo = new short[((short) (7 * 3))];// Contains RAM and EEPROM r
4      m_memoryInfoOffset = snapshotAvailableMemory(((short) (1)), m_memoryInfo
5      if (bTEST_256b_CURVE) {
6          m_ecc = new ECConfig(((short) (256)));
7      }
8      if (bTEST_512b_CURVE) {
9          m_ecc = new ECConfig(((short) (512)));
10     }
11     m_memoryInfoOffset = snapshotAvailableMemory(((short) (2)), m_memoryInfo
12     // Pre-allocate test objects (no new allocation for every tested operat
13     if (bTEST_256b_CURVE) {
14         m_testCurve = new ECCurve(false, SecP256r1.p, SecP256r1.a, SecP256r1
15         m_memoryInfoOffset = snapshotAvailableMemory(((short) (3)), m_memory
16         // m_testCurveCustom and m_testPointCustom will have G occasionally
17         m_customG = new byte[((short) (SecP256r1.G.length))];
18         Util.arrayCopyNonAtomic(SecP256r1.G, ((short) (0)), m_customG, ((sho
19         m_testCurveCustom = new ECCurve(false, SecP256r1.p, SecP256r1.a, Sec
20     }
21     if (bTEST_512b_CURVE) {
```



Checking for non-constant time execution

opencrypto.jcmathlib.OCUnitTests#test_BN_MOD(javacard.framework.APDU,short)

TRAP_opencrypto_jcmathlib_OCUnitTests_hash_test_BN_MOD_argb_javacard_framework_APDU__short_arge_10

Mode: time
Card ATR: 3B80800101
Number of rounds: 1000
APDU header: B0252100
Input regex: 00[0-9A-F]{64}[4-7][0-9A-F]{63}
Elapsed time: 0 days 00:58:39.803
Source measurements: measurements.csv

Show explicit traps

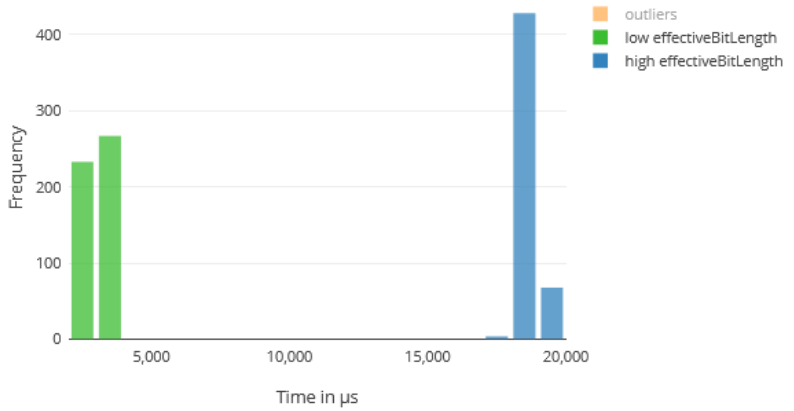
Avg µs

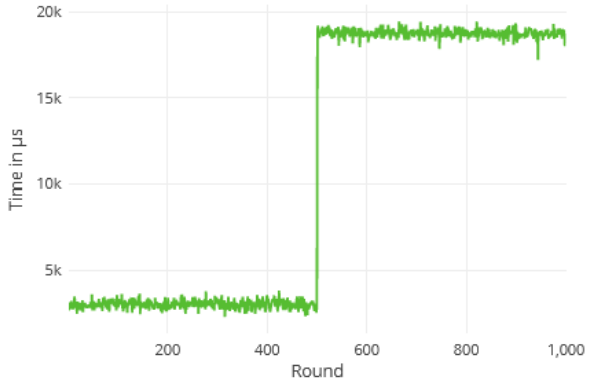
```
1  void test_BN_MOD(APDU apdu, short dataLen) {
2      byte[] apdubuf = apdu.getBuffer();
3      short p1 = ((short) (apdubuf[ISO7816.OFFSET_P1] & 0xff));
4      Bignat num = m_testBN1;
5      num.set_size(p1);
6      Bignat mod = m_testBN2;
7      mod.set_size(((short) (dataLen - p1)));
8      num.from_byte_array(p1, ((short) (0)), apdubuf, ISO7816.OFFSET_CDATA);
9      mod.from_byte_array(((short) (dataLen - p1)), ((short) (0)), apdubuf,
10     num.mod(mod);
11     short len = num.copy_to_buffer(apdubuf, ((short) (0)));
12     apdu.setOutgoingAndSend(((short) (0)), len);
13 }
```

Colour explanation

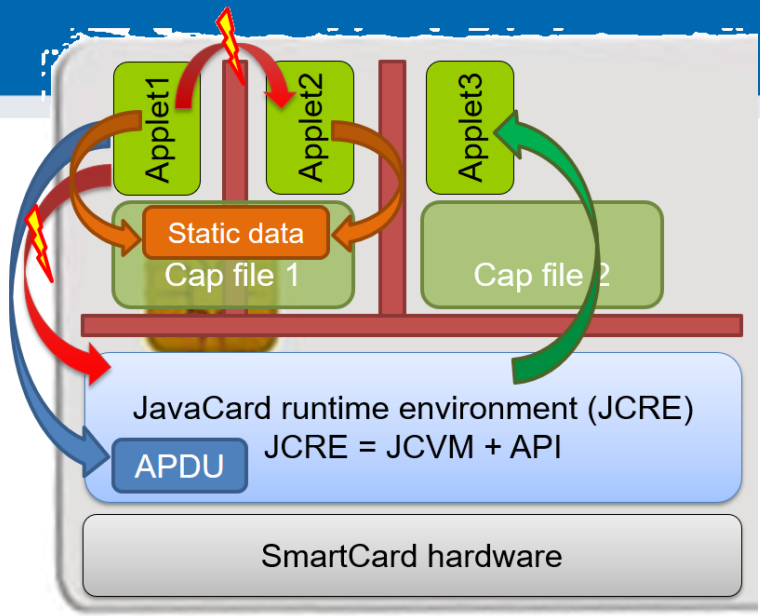
Currently selected trap | Trap was never reached | Trap was reached only sometimes

Click on a graph item to get a list of corresponding inputs.





JavaCard applet firewall issues



- Main defense for separation of multiple applets
- Platform implementations differ
  - Usually due to the unclear and complex specification
- If problem exists, then is out of developer's control
- Firewall Tester project (W. Mostowski)
  - Open and free, the goal is to test the platform
  - http://www.sos.cs.ru.nl/applications/smartcards/firewalltester/

```
short[] array1, array2; // persistent variables
short[] localArray = null; // local array
JCSystem.beginTransaction();
        array1 = new short[1];
        array2 = localArray = array1; // dangling reference!
JCSystem.abortTransaction();
```

Relevant open-source projects

- Easy building of applets
  - https://github.com/martinpaljak/ant-javacard
  - https://github.com/ph4r05/javacard-gradle-template
- AppletPlayground (ready to “fiddle” with applets)
  - https://github.com/martinpaljak/AppletPlayground
- Card simulator https://jcardsim.org
- Profiling performance
  - https://github.com/crocs-muni/JCAlgTest
  - https://github.com/lzaoral/JCProfilerNext
- Curated list of JavaCard applets
  - https://github.com/crocs-muni/javacard-curated-list
- Low-level ECPoint library
  - https://github.com/OpenCryptoProject/JCMathLib

THRESHOLD CRYPTOGRAPHY (TO REMOVE SINGLE POINT OF FAILURE)

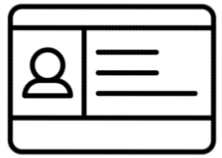
Possibly heard of ROCA vulnerability CVE-2017-15361

M. Nemec, M. Sys, P. Svenda, D. Klinec, V. Matyas: The Return of Coppersmith's Attack..., ACM CCS 2017

The usage domains affected by the vulnerable library

Austria, Estonia, Slovakia, Spain…

Identity documents (eID, eHealth cards)

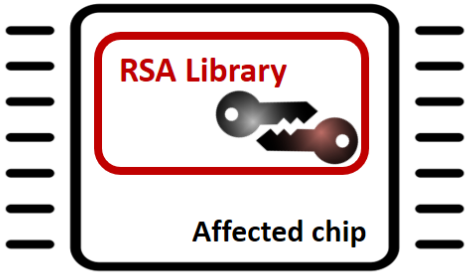


Trusted Platform Modules (Data encryption, Platform integrity)

25-30% TPMs worldwide, BitLocker, ChromeOS… Firmware update available

Software signing

Commit signing, Application signing GitHub, Maven…



Secure browsing (TLS/HTTPS*)

Very few keys, but all tied to SCADA management

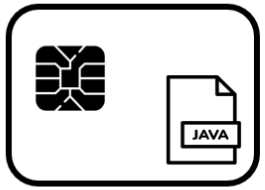
Authentication tokens

Gemalto .NET Yubikey 4…

Message protection (S-MIME/PGP)

Yubikey 4…

Programmable smartca

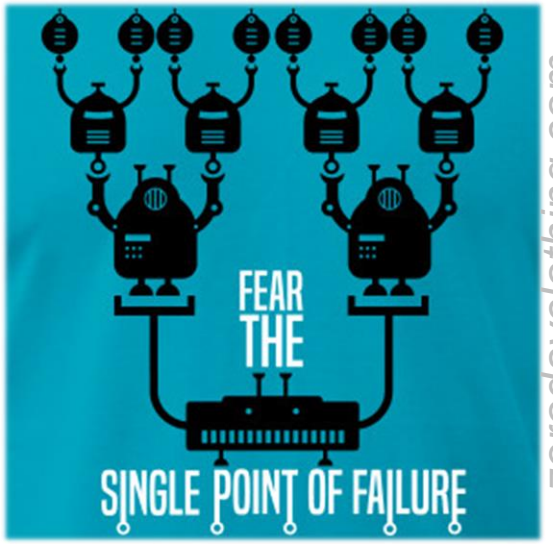


a small number of vulnerable ke

Single point of failure: Prime generation of RSA keygen in widely used chip (1-2 billion chips)

Single point of failure

- We already try to avoid single point of failure at many places
  - Personal: dual control, people from different backgrounds…
  - Technical: Load-balancing web servers, RAID, periodic backups…
  - Supply chain: no reliance on single supplier…
- Problems: Appropriate trade-off between security, cost and usability
- Systems without single point of failure tend to be:
  - More complex
  - More expensive
  - Less performant
  - Backward incompatible
  - (not really without single point of failure)



zerodayclothing.com

REMOVING SINGLE POINT OF FAILURE IN CRYPTOGRAPHIC SIGNATURES

Focus of this lecture (threshold cryptography)

Secure Multi-Party Computation

- "Offload heavy computation to untrusted party while not leaking info"

Example:

- Amazon evaluates trained neural network on medical image (on behalf of user)
- Amazon learns neither the trained NN, nor the processed image
- *Technology*: Homomorphic encryption, garbled circuits (slow, but getting better)

- "Distribute critical cryptographic operation among N parties"

Example:

- 3 devices collaboratively compute digital ECC signature
- Private key never at single place, secure unless all devices are compromised
- Technology: purpose tailored schemes (efficient, provably secure)

Goals of threshold cryptography

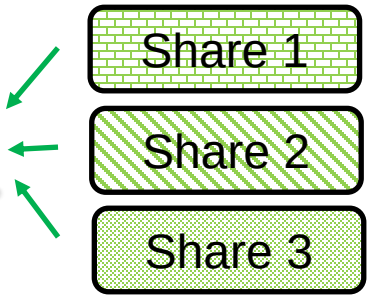
1. Remove single point of failure
   - Reduce trust requirements (no single party can fail you)
   - Better protect against attacks like malware (no single device with full key)
   - Provide resiliency against subset compromise (k-of-n)
2. Provide fault tolerance (n-of-n vs. k-of-n)
   - Perform operation with some parties not available
3. Enable different signing/decryption policies to be enforced (each party)
   - Regulatory requirements (e.g., “four eyes principle”)

Analogically for decryption (single person decrypts, multiple people, k-of-n)

Threshold Signature (MPC)

Single signature

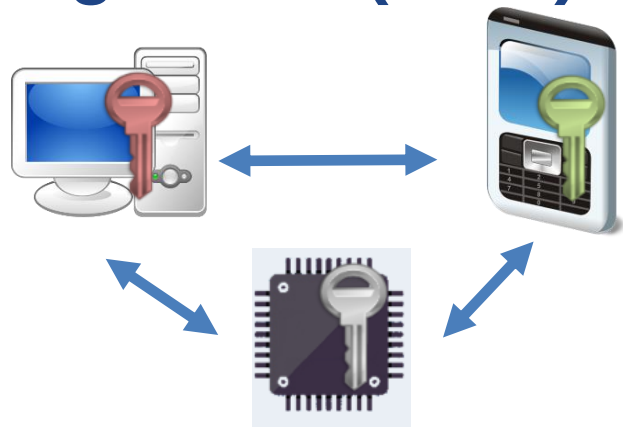
Shamir TSS



Signature

Multiple signatures

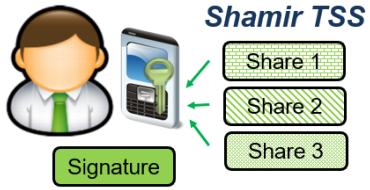
Signature | Signature | Signature



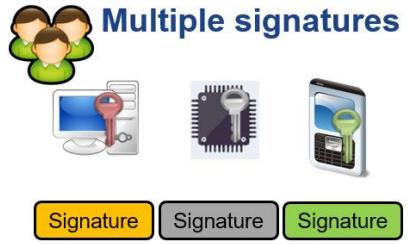
Signature

How to “split” key for threshold cryptography?

- (Shamir threshold secret sharing)

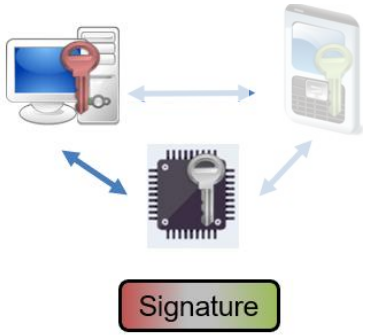


- Multiple separate signatures (same algorithm)



- Cryptographic “garden” (multiple keys, different keys)

- n-of-n MPC signature (multiple keys, all must contribute)
  - k-of-n MPC threshold signature (multiple keys, k must contribute)



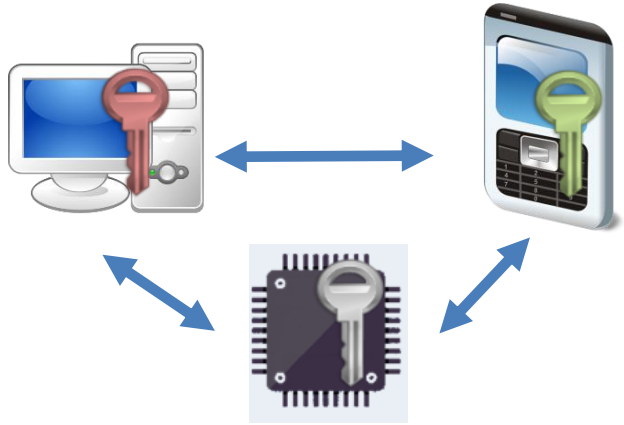
Option: Cryptographic “garden”

- Electronic signature == sign_RSA(SHA256(message))
  - Failure in RSA or SHA256 algorithm or its implementation => forgery of signatures

- Signature using cryptographic “garden”
  - Differently computed (algorithm) signatures over same message
  - Signature = sign_RSA+ sign_ECC + sign_PostQuantumAlg
  - Mitigate design problems of particular algorithm

- Disadvantages: backward (in-)compatibility, larger storage space…

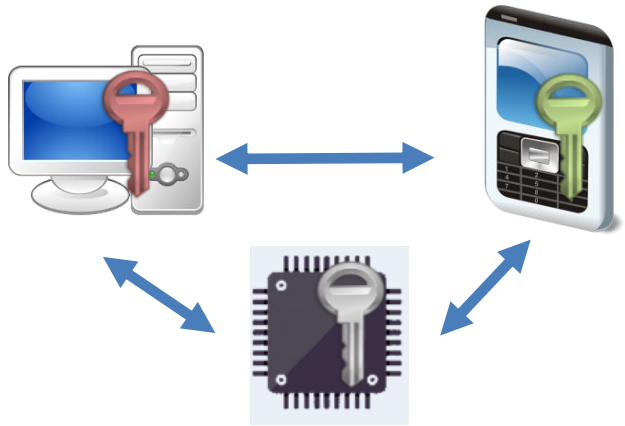
Threshold cryptography



- Proposed already in 1987 by Y. Desmedt
- Principle
  - Private key split into multiple parts (“shares”)
  - Shares used (independently) by separate parties during a protocol to perform desired cryptographic operation
  - If enough shares are available, operation is finished successfully
- Properties
  - Better protection of private key (single point of failure removed)
  - Key shares can be distributed to multiple parties (independent usage condition)
  - Resulting signature may be indistinguishable from a standard one (e.g., ECDSA)
- Significant research progress made in the cryptocurrency context

Threshold cryptography protocols

- Typically, distributed key generation is also included
  - Private key is not generated on a single device
- Output signatures can be indistinguishable from single party signatures
  - ECDSA ([GGN16], [LN18], [GG18], [GG20], [Can+20], …)
  - Schnorr (MuSig, MuSig2, FROST…)
  - RSA ([DF91], [Gen+97], [DK01], Smart-ID…)
- Various designs with different properties
  - Supported setups (n-of-n / k-of-n)
  - Number of communication rounds
  - Computation complexity
  - Security assumptions…



PRACTICAL EXAMPLES OF MPC

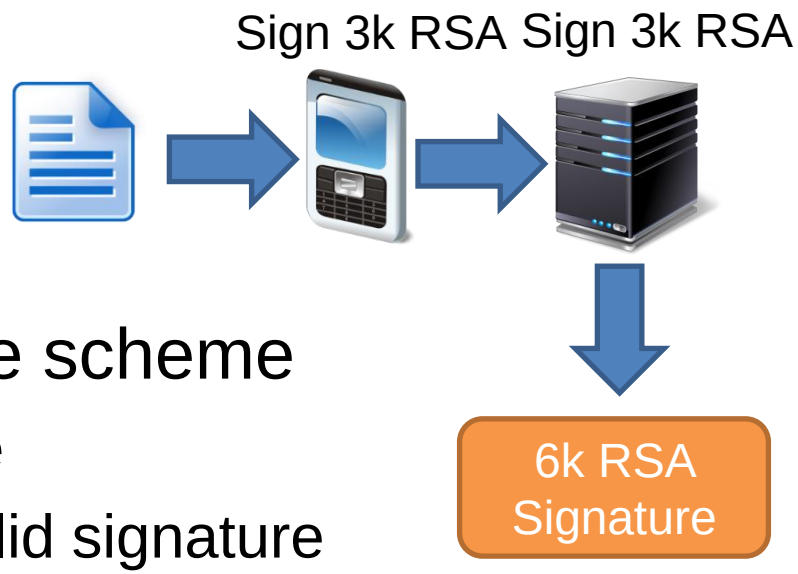
Smart-ID signature system

Server-Supported RSA Signatures for Mobile Devices

Ahto Buldas[1,2(✉)], Aivo Kalu[1], Peeter Laud[1], and Mart Oruaas[1]

[1] Cybernetica AS, Tallinn, Estonia
ahto.buldas@cyber.ee
[2] Tallinn University of Technology, Tallinn, Estonia

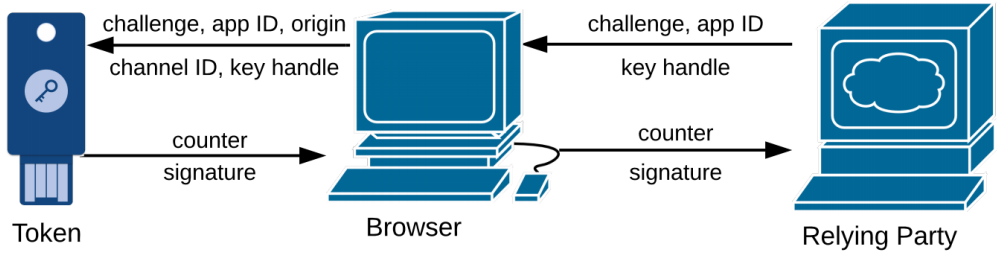
- Banks in Baltic states, >4M users
  - Qualified Signature Creation Device (QSCD)!
- Collaborative computation of signature using:
  1. User's mobile device (3072b RSA)
  2. Smart-ID service provider (3072b RSA)
- Two-party RSA signatures, threshold signature scheme
  - Whole signature key never present at a single place
  - Smart-ID service provider cannot alone compute valid signature
- Final signature is 6144b RSA => compatible with existing systems
  - Assumed security level is equivalent to 3072b RSA (as if one party compromised)



MPC wallets (software, hardware)

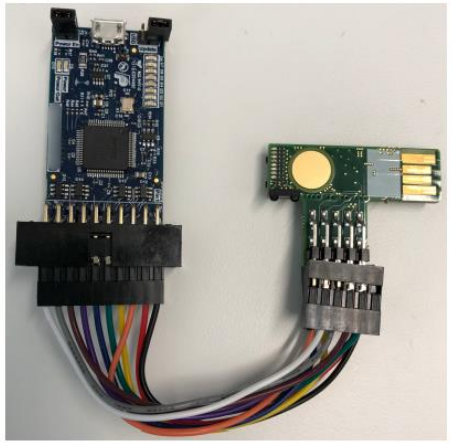
- Number of cryptocurrencies uses ECDSA/EdDSA/Schnorr algorithm to authorize TX
  - Funds are lost if private key is stolen/lost
- Multiple separate signatures by separate private keys possible (so called multisignature)
  - More costly (more onchain space => higher fee)
  - Privacy leaking (structure of approval)
  - Not always (directly) supported (Bitcoin has IP_CHECKMULTISIG, Ethereum needs special contract)
- MPC to compute threshold multiparty signature
  - Interaction between multiple entities, single signature as a result
  - Not recognizable from standard transactions on-chain
- ECDSA
  - Several end-user wallets like ZenGo, Binance, Coinbase… as well as institutional custodians
  - Usually one share by user, second by server
- Schnorr-based signatures easier to compute (e.g., Musig-2, FROST)
  - Available in Bitcoin after Taproot

True2F FIDO U2F token



- Yubikey 4 has single master key
  - To efficiently derive keypairs for separate Relying parties (Google, GitHub…)
  - Inserted during manufacturing phase (what if compromised?)
- Additional two MPC protocols (as protection against backdoored token)
  - Verifiable insertion of browser randomness into final keypairs
  - Prevention of private key leakage via ECDSA padding

- Backward-compatible (Relying party, HW)
- Efficient: 57ms vs. 23ms to authenticate



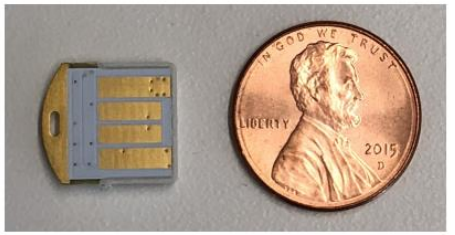
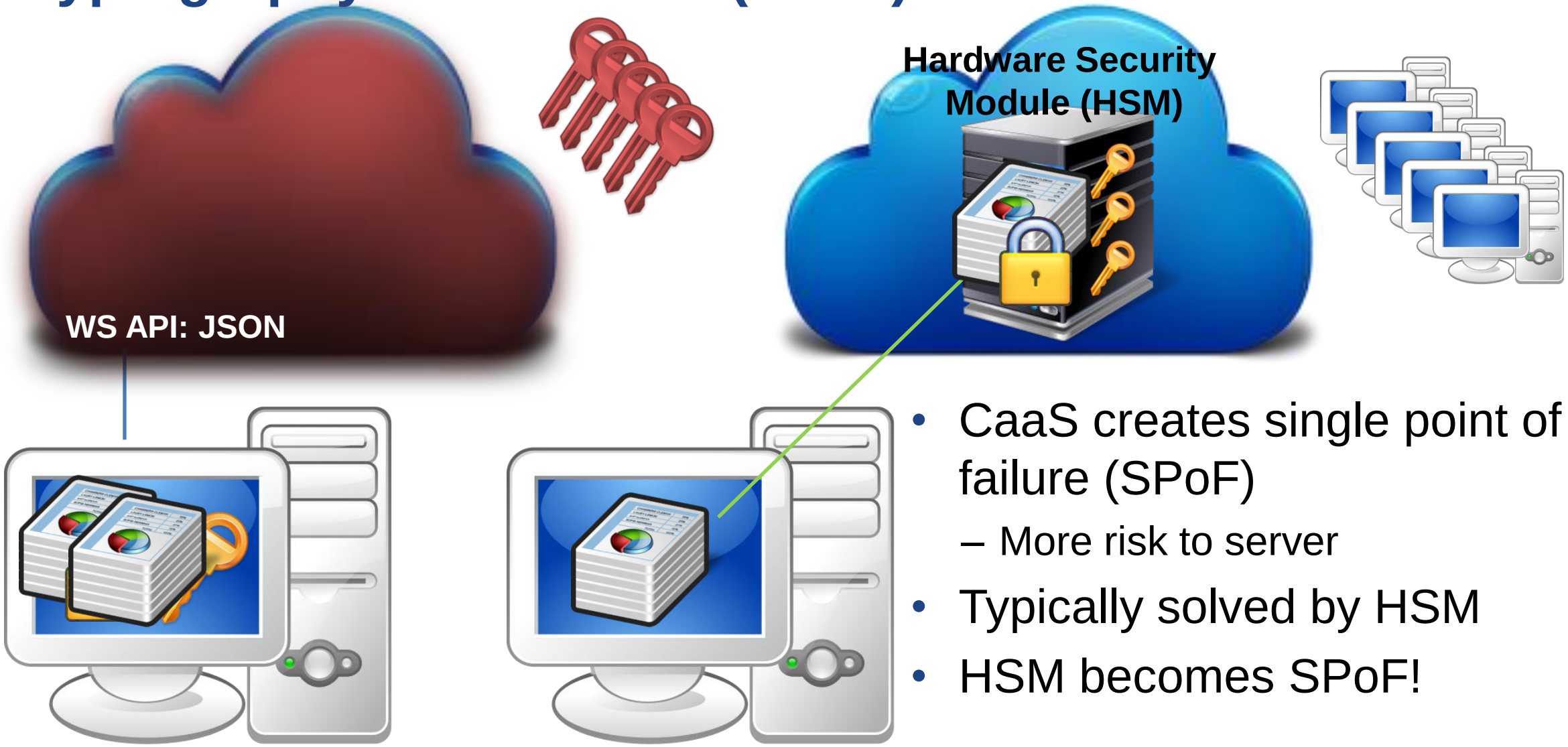


Figure 1: Development board used to evaluate our True2F prototype (at left) and a production USB token that runs True2F (above).

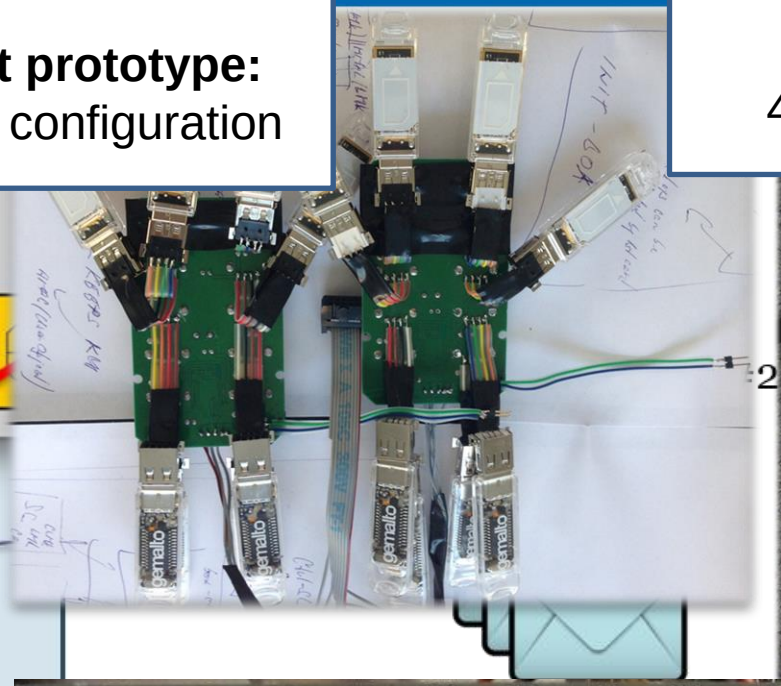
Cryptography as a Service (CaaS)



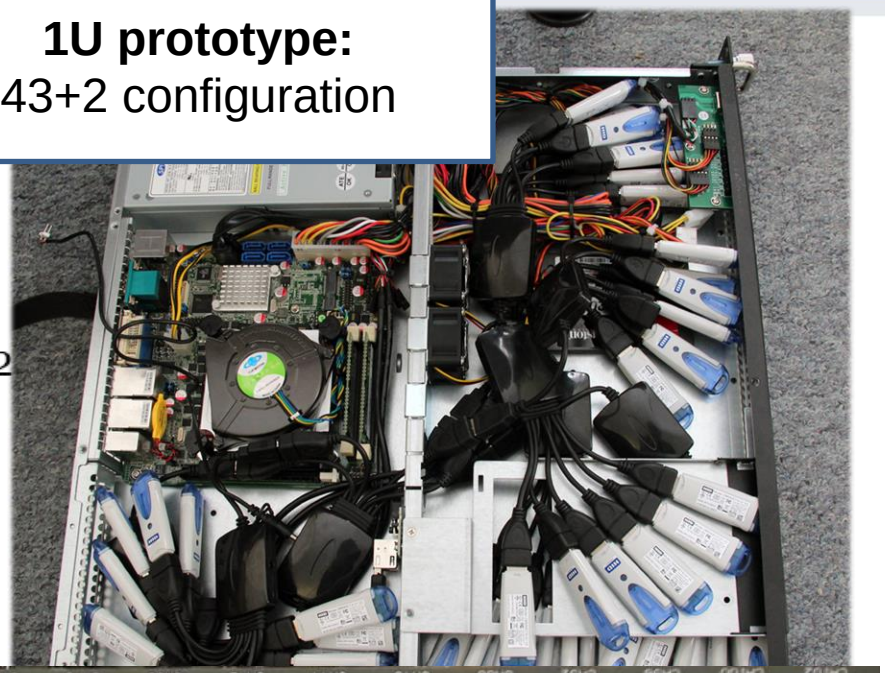
- CaaS creates single point of failure (SPoF)
  - More risk to server
- Typically solved by HSM
- HSM becomes SPoF!

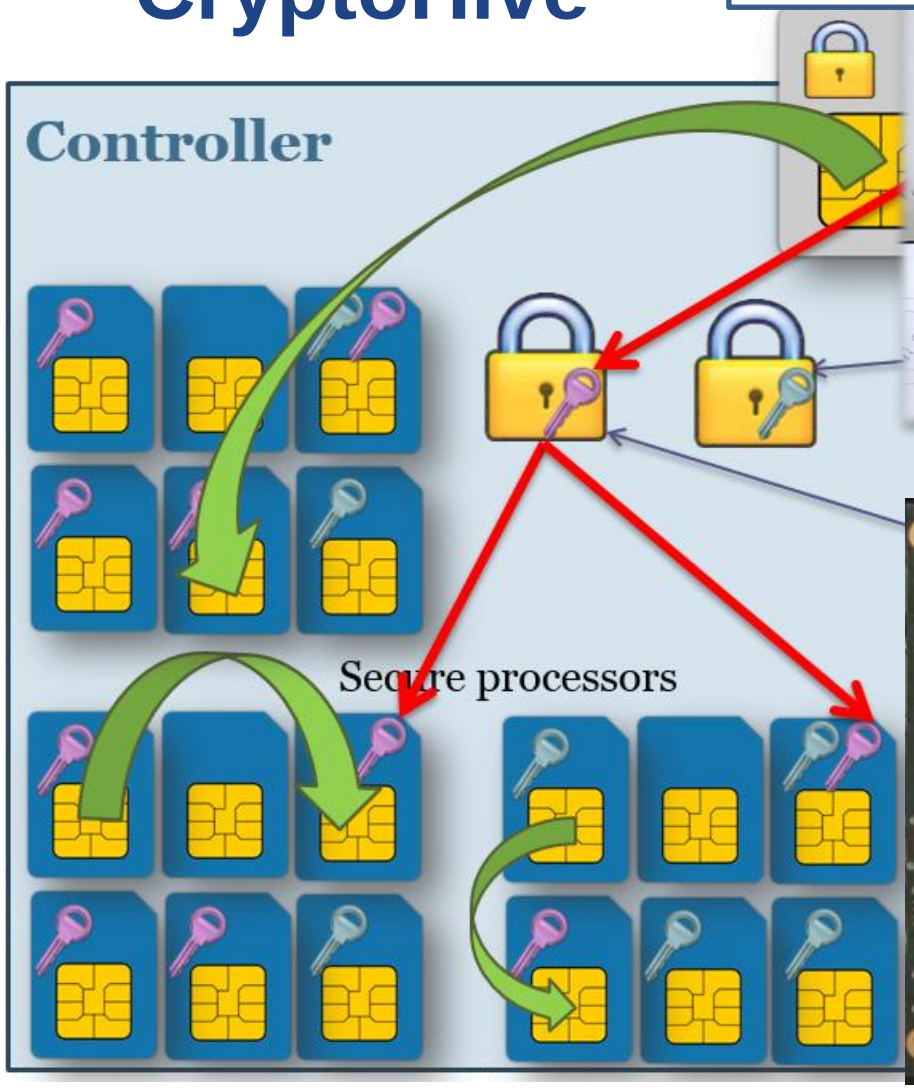
CryptoHive

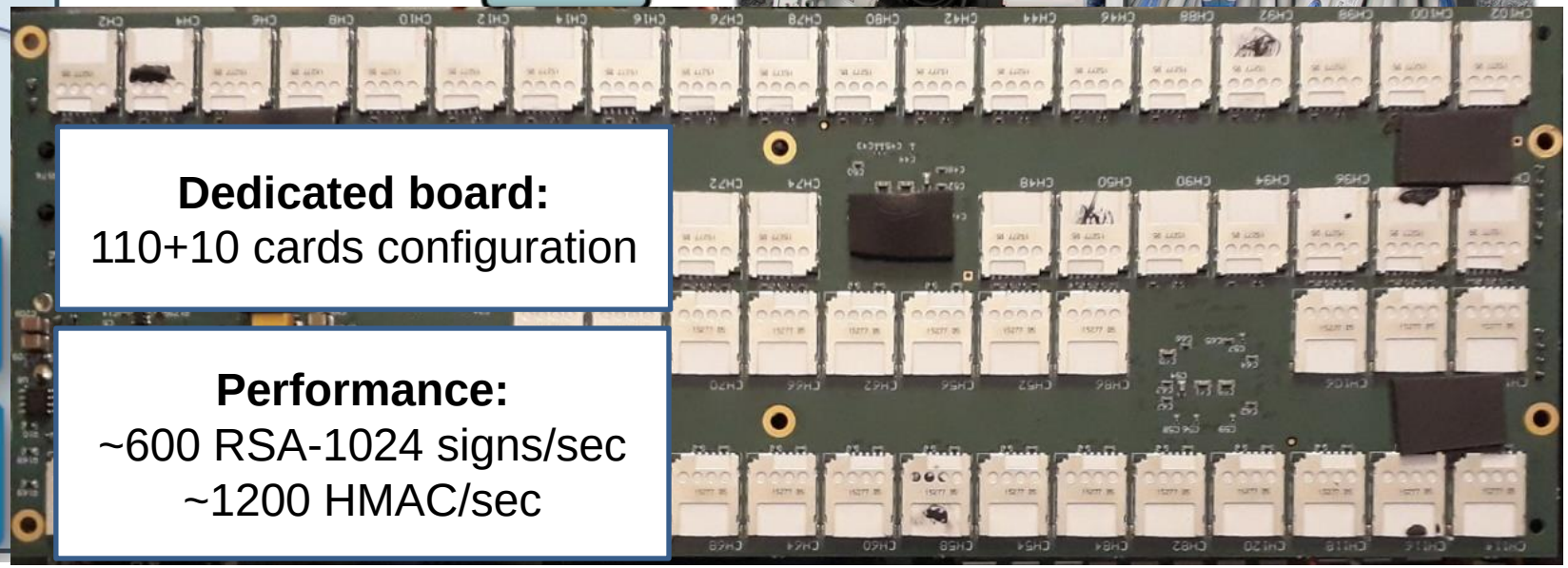
First prototype:
12+2 configuration



1U prototype:
43+2 configuration



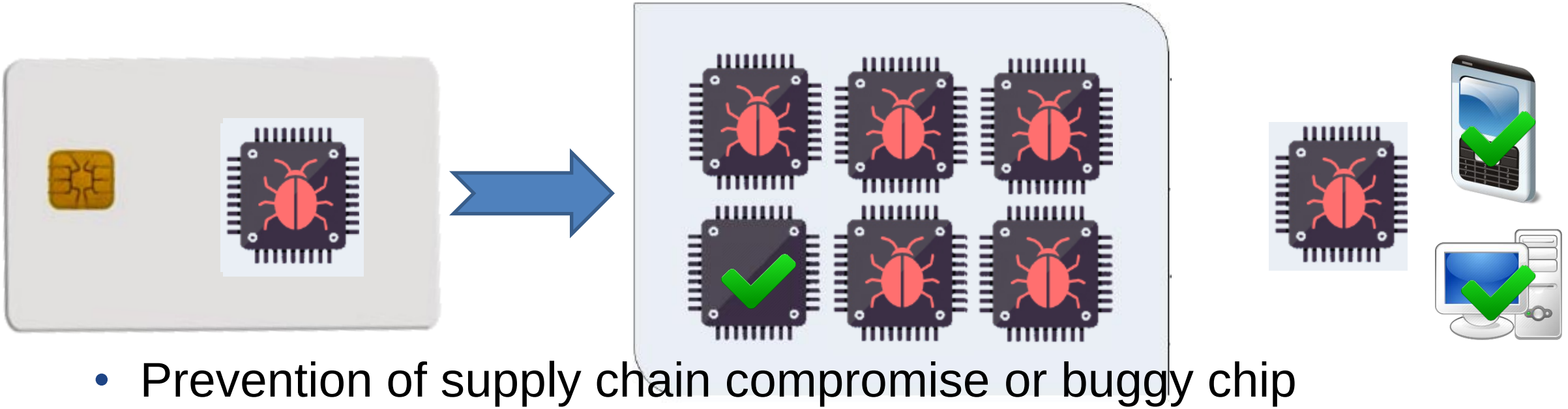




Dedicated board:
110+10 cards configuration

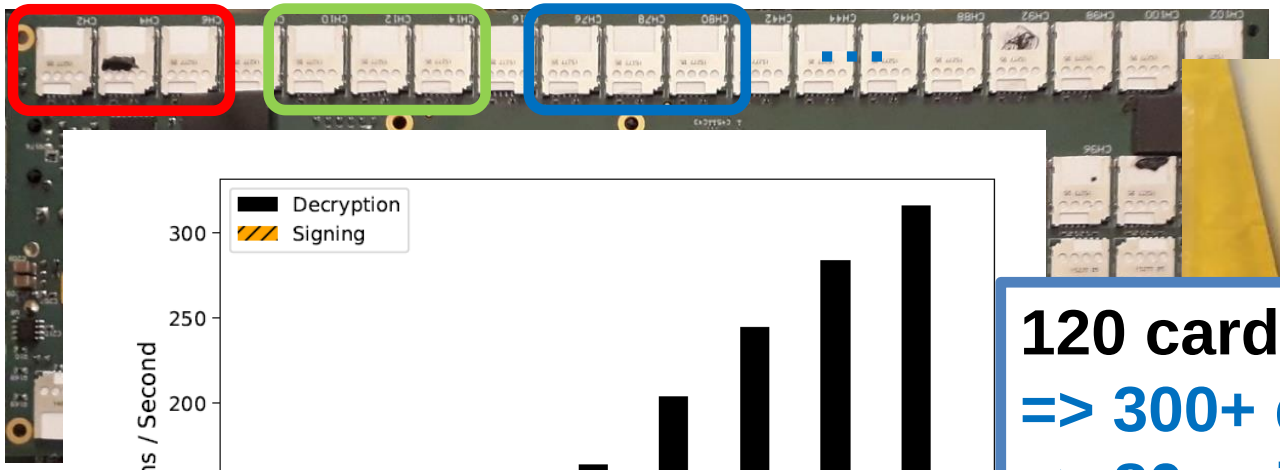
Performance:
~600 RSA-1024 signs/sec
~1200 HMAC/sec

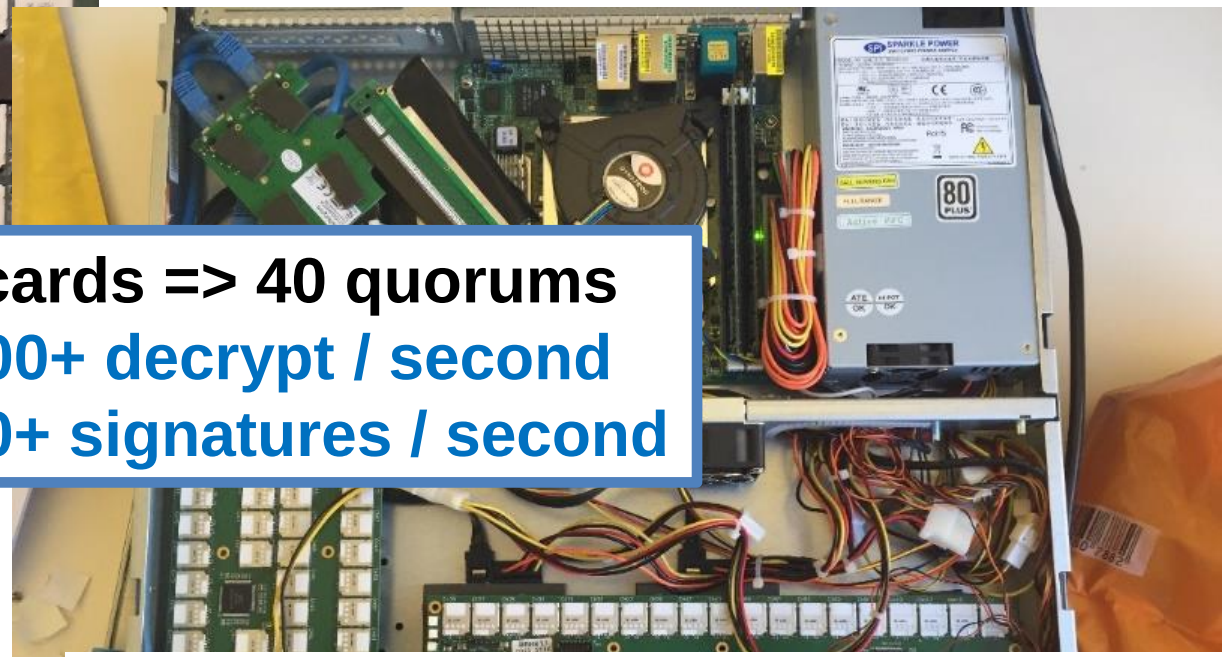
Problem: buggy or subverted chip



- Prevention of supply chain compromise or buggy chip
- Suite of ECC-based multi-party protocols proposed
  - Distributed key generation, ElGamal decryption, Schnorr signing
- Efficient implementation on JavaCards + high-speed box
- Combination with non-smartcard devices possible

SmartHSM for multiparty (120 smartcards, 3 cards/quorum)





120 cards => 40 quorums
=> 300+ decrypt / second
=> 80+ signatures / second

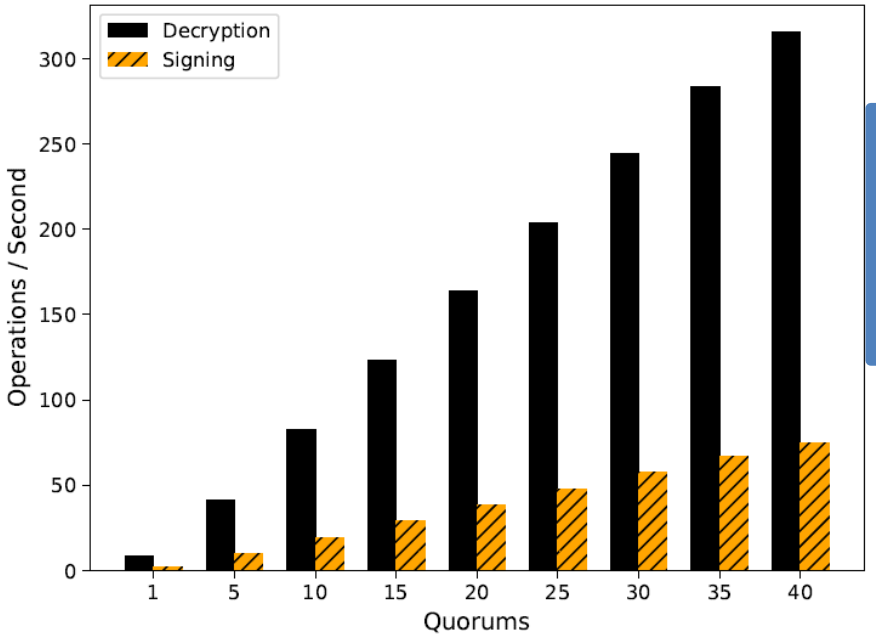


Figure 10: The average system throughput in relation to the number of quorums ($k = 3$) that serve requests simultaneously. The higher is better.

A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components

Vasilios Mavroudis
University College London
v.mavroudis@cs.ucl.ac.uk

Andrea Cerulli
University College London
andrea.cerulli.13@ucl.ac.uk

Petr Svenda
Masaryk University
svenda@fi.muni.cz

Dan Cvrcek
EnigmaBridge
dan@enigmabridge.com

Dusan Klinec
EnigmaBridge
dusan@enigmabridge.com

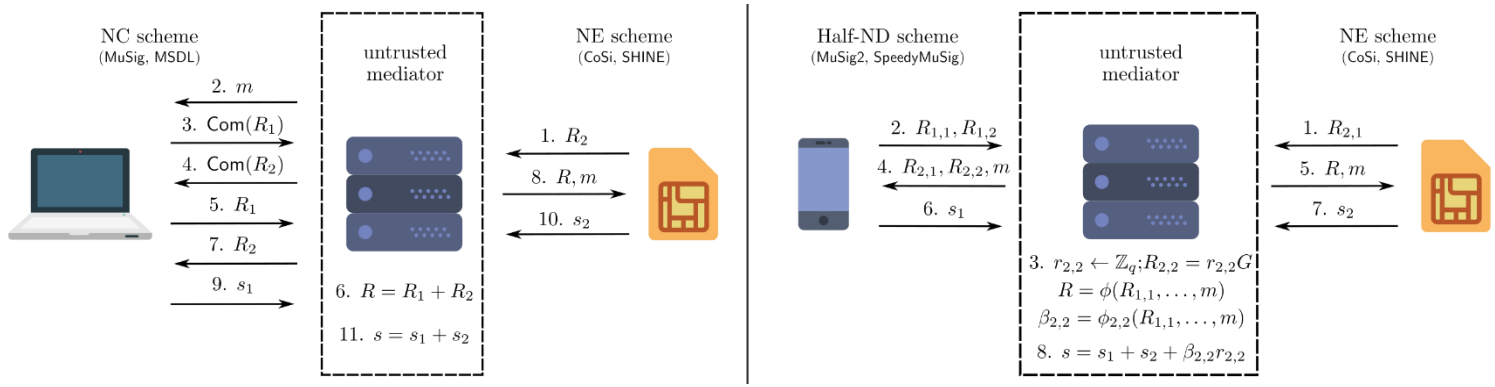
George Danezis
University College London
g.danezis@ucl.ac.uk

How to run MPC on JavaCards

- MPC applets: https://github.com/OpenCryptoProject/Myst, https://github.com/crocs-muni/JCFROST
- Schnorr-based MPC protocols requires low-level curve operations
  - Supported by card, but not exposed by standard JavaCard API
- JCMathLib https://github.com/OpenCryptoProject/JCMathLib
  - Adds support for low-level classes/methods like ECPoint and Integer
    - Which are otherwise not supported by public JavaCard API
    - (available via proprietary extensions, but requires NDA)
  - Main goals
    1. Expose helpful functions for research/FOSS usage (e.g., Schnorr MPC sigs)
    2. Allow for publication of functional applets originally based on proprietary API
  - Low-level methods build (mis)using existing JC API
    - E.g., ECPoint.multiply() using ECDH KeyAgreement + additional computation
  - Optimized for low RAM memory footprint and performance

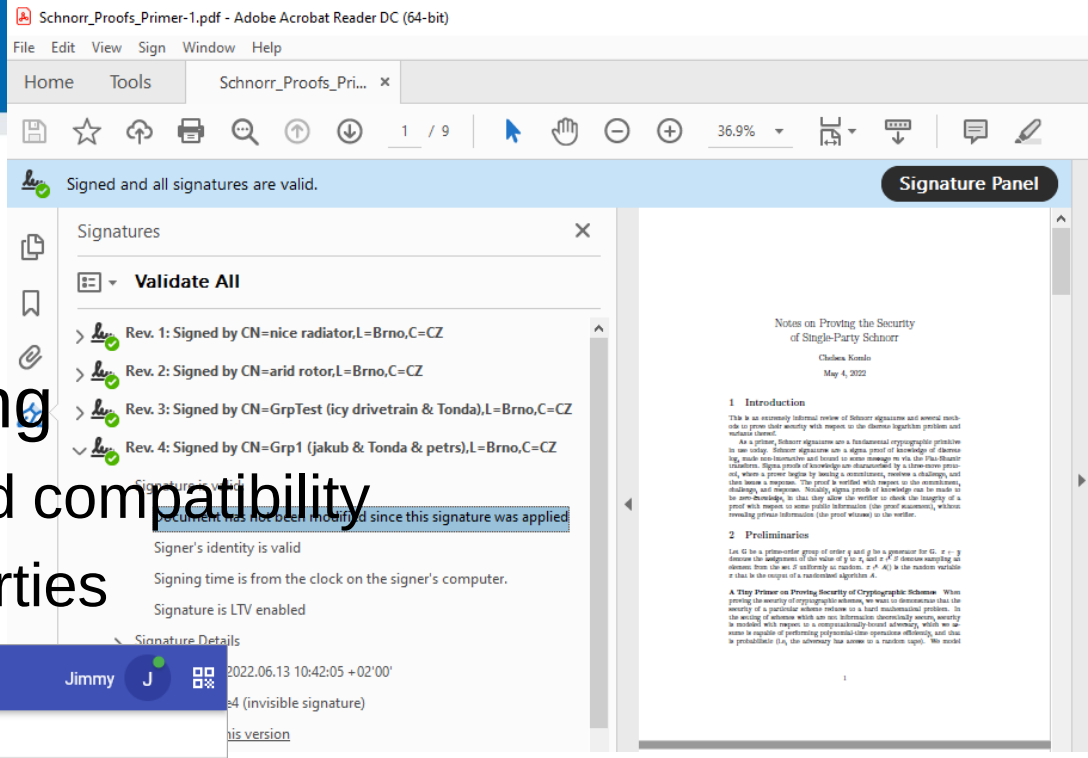
SHINE: Interoperability of MPC signatures

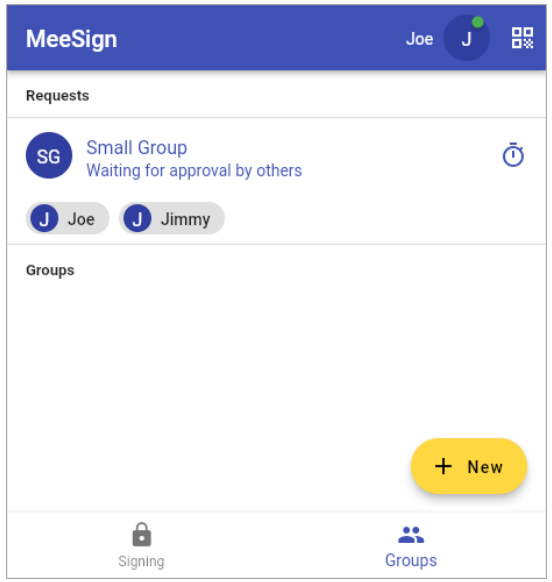
- Idea: make existing Schnorr-based MPC protocols interoperable via untrusted mediator
  - NE-based schemes (CoSi, Myst)
  - NC-based schemes (MuSig, MSDL)
  - Half-ND-based schemes (MuSig2, SpeedyMuSig)
- Additional multi-signature protocol optimized for smartcards (SHINE)
  - JCMathLib used

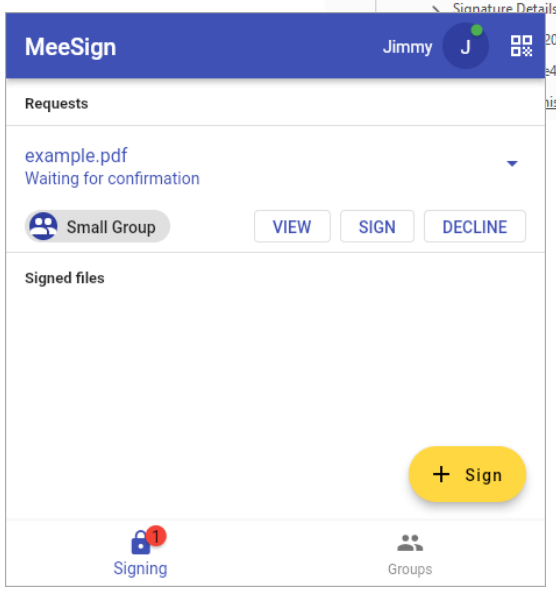


MeeSign (k-of-n ECDSA)

- Platform for multi-party document signing
  - MPC utilized for group signing and backward compatibility
  - Outputs valid PDF signatures by multiple parties



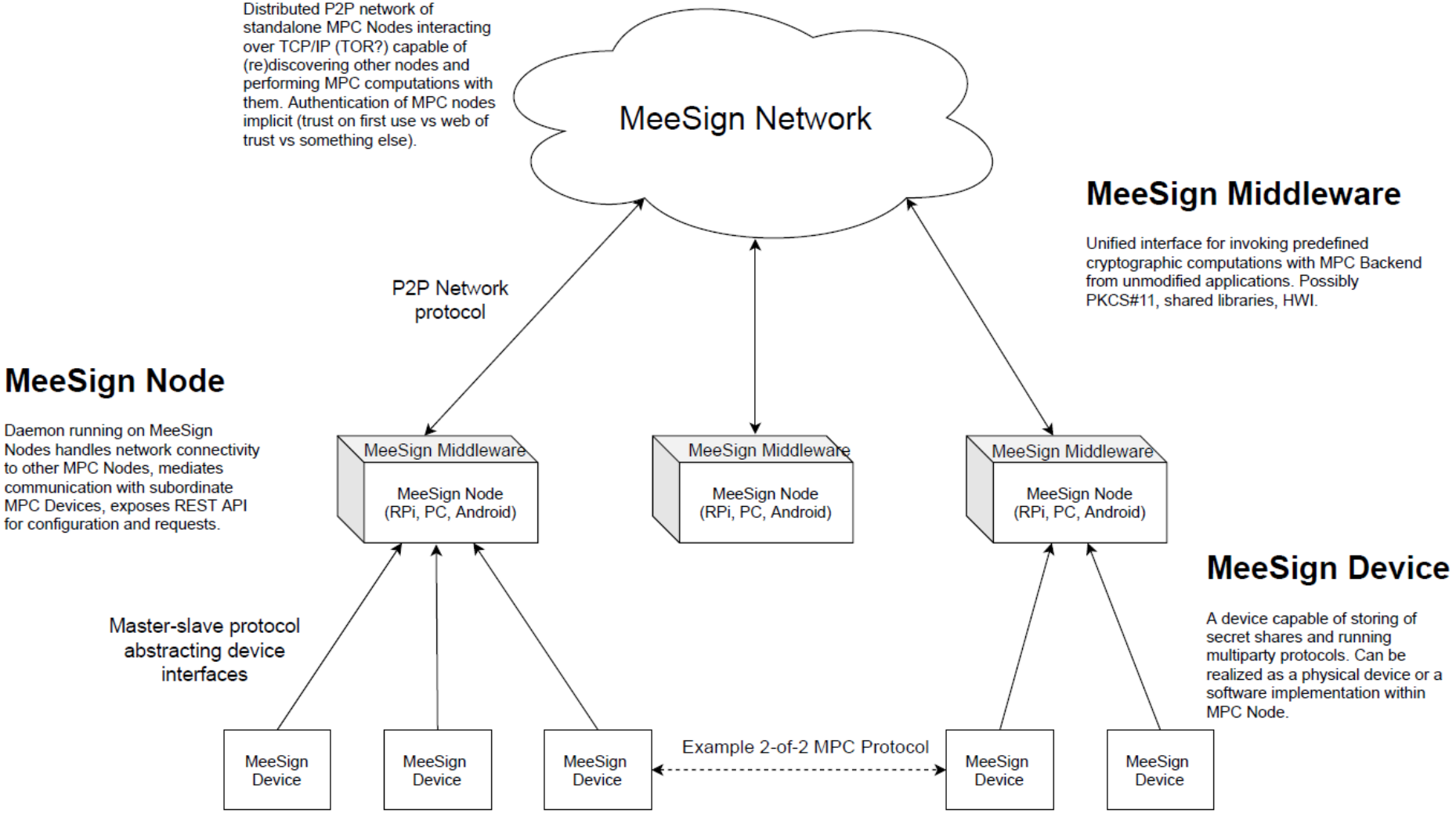




MeeSign: Threshold signing for electronic evidence management. Antonin Dufka, Jakub Janku, Jiri Gavenda, Petr Svenda. EurOpen 2022. Also available on https://meesign.crocs.fi.muni.cz/.

MeeSign Network

Distributed P2P network of standalone MPC Nodes interacting over TCP/IP (TOR?) capable of (re)discovering other nodes and performing MPC computations with them. Authentication of MPC nodes implicit (trust on first use vs web of trust vs something else).



MeeSign Middleware

Unified interface for invoking predefined cryptographic computations with MPC Backend from unmodified applications. Possibly PKCS#11, shared libraries, HWI.

MeeSign Node

Daemon running on MeeSign Nodes handles network connectivity to other MPC Nodes, mediates communication with subordinate MPC Devices, exposes REST API for configuration and requests.

MeeSign Device

A device capable of storing of secret shares and running multiparty protocols. Can be realized as a physical device or a software implementation within MPC Node.

MeeSign Protocols

Various MPC protocols for signing, decryption, randomness generation (possibly generic?) running on MPC Devices and controlled by MPC Nodes

USE-CASE SCENARIOS

High-level usage scenarios

1. Digital signature
2. User authentication
3. Data decryption
4. Key / randomness generation

Multiparty signatures – configurations and use-cases

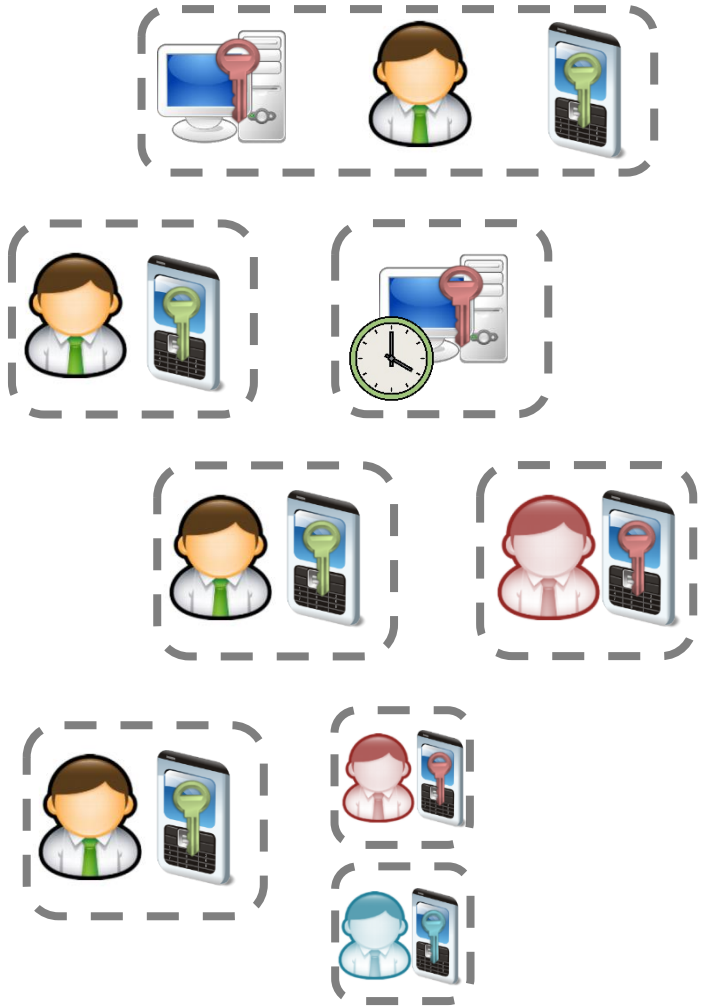
- 2-out-of-2 (two signers, both required)
  - One share on mobile phone, second on server (Smart-ID, eIDAS compliant)
  - One share on US smartcard, second on Chinese smartcard (backdoor resistance)
- 2-out-of-3 (three signers total, at least two required)
  - Two shares user, one share backup server (backup if user lose one share)
  - One share lender, one share lendee, one share arbiter (for disputes)
- 1-out-of-3 (very robust backup against key loss)
- 3-out-of-5 (shares distribution voting)
  - CEO has 2 shares, all other have only single one
- 11-out-of-15 (Liquid consortium signing blocks on Liquid sidechain)

Multiparty signatures with additional policy

- Signers can also enforce specific signing policy
  - Only during certain time, documents, type of operations, certain amount…
- 2-out-of-2 with policy
  - One person, second automatic signer only during office hours
- 2-out-of-3 with policy (two people, one automated device with policy)
  - Two people together can always sign/transfer, one person alone only up to limit)
- 3-out-of-3 (two people, one automated device with policy)
  - Automated device signs only when previous two already signed and additionally impose 1-month delay (timelock)

MPC for authentication – configurations

- 2-of-2: one user, two devices
  - (higher security against device compromise)
- 2-of-2: one user, one server-side automatic process
  - (check time interval when authentication is allowed)
- 2-of-2: two users (user, approving controller)
  - (access must be approved by controller)
- 2-of-3: three users (user, redundant approvers)
  - (one user, two controllers – one approval is enough)
- Bonus: Independent log of authentication attempt



Multiparty decryption and Shamir threshold scheme

- Combination of MPC and Shamir
  - 2-of-2 multiparty decryption for every person to decrypt Shamir share
  - Shamir shares combined later (standard procedure)
  - Usable to enable easy removal of person from share (by deletion of second key for 2-of-2)

Threshold crypto protocols – tradeoffs and limitations

- Security vs. usability
- More difficult to finalize signature (more parties, communication)
- More complex software (bugs more likely)
- Number of rounds, interaction
- Amount of data exchanged
- Active research field => possibility for new attacks against whole schemes

Backward compatibility

- Existing systems already use some crypto algorithm (RSA, ECDSA…)
  - Difficult to switch all information systems to new algorithm
- Threshold algorithm is backward compatible, if verification is unchanged
  - Only signer needs to update its software (to create threshold MPC signature)
  - Verification software stay unchanged
  - Allows for gradual opt-in (improve signing security of people who upgrade)
  - (similarly for decryption – if encryption is unchanged)
- Backward-compatible signatures exists for commonly used algorithms
  - RSA, ECDSA, EdDSA, Schnorr…

Summary

- JavaCard programming
  - Optimizations need to consider underlaying hardware (RAM, co-processors…)
  - Programs shall anticipate faults during computation (injected by an attacker)
- Secure Multiparty Computation
  - Exciting domain, active research, many practical uses
  - Collaborative computation of signatures, decryption, keygen…
  - Can be backward compatible (k-ECDSA, k-RSA, k-Schnorr…)
  - Usually more computational demanding (common CPU is enough)
  - Some protocols efficient enough to run on smartcards (Schnorr-based sigs…)
- Split to multiple parties provides:
  - Better protection of private key against bugs and compromise
  - Possibility of additional policy before party participation

Additional slides for generic multiparty computation and whitebox cryptography construction (for interested, not mandatory part of PV204 course)

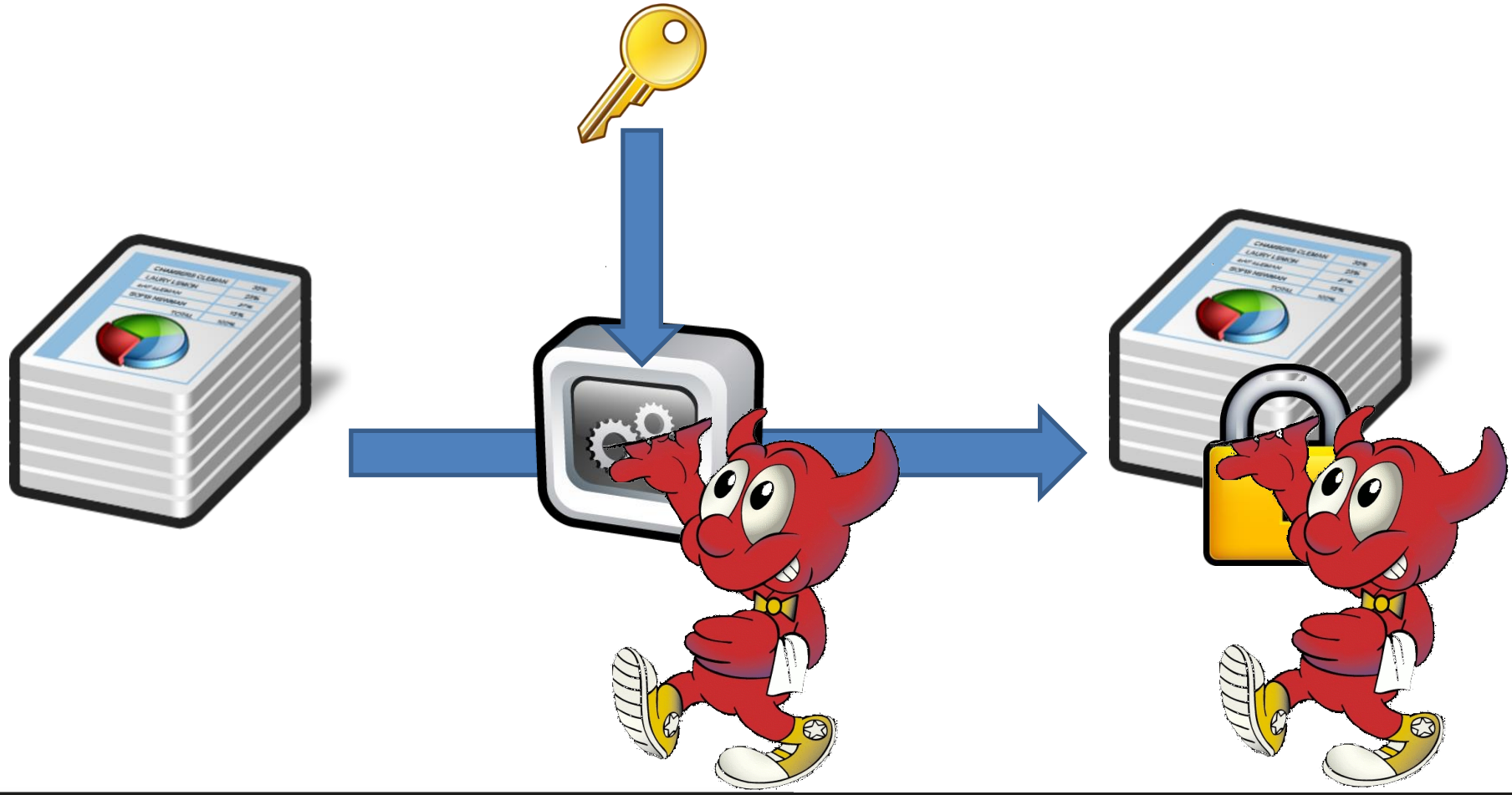
Secure Multiparty Computation (MPC)

- Enables multiple parties to jointly compute a function, but keeps their inputs private
- Different types of protocols:
- Secret sharing protocols (e.g., Shamir)
  - Jointly hold secret, but split in shares, can be reconstructed only when sufficient number of shares are available
- Garbled circuit protocol
  - One party creates Boolean circuit (2-input gates) from the specification and transforms it (XOR, AND)
  - Other party evaluates without learning original specification and/or data
- Partial/Somewhat/Fully Homomorphic Encryption (PHE, SHE, FHE)
- Oblivious Transfer, Secure Function Evaluation

Protections Against Reverse Engineering

HOW TO PROTECT

Standard vs. whitebox attacker model (symmetric crypto example)



Whitebox attacker model

- The attacker is able to:
  - inspect and disassemble binary (static strings, code...)
  - observe/modify all executed instructions (OllyDbg...)
  - observe/modify used memory (OllyDbg, memory dump...)
- How to still protect value of cryptographic key?
- Who might be whitebox attacker?
  - Mathematician (for fun)
  - Security researcher / Malware analyst (for work)
  - DRM cracker (for fun&profit)
  - ...

Classical obfuscation and its limits

- Provides only time-limited protection
- Obfuscation is mostly based on obscurity
  - add bogus jumps
  - reorder related memory blocks
  - transform code into equivalent one, but less readable
  - pack binary into randomized virtual machine...
- Barak's (im)possibility result (2001)
  - family of functions that will always leak some information
  - but practical implementation may exist for others
- Cannetti et. al. positive results for point functions
- Goldwasser et. al. negative result for auxiliary inputs

Computation with Encrypted Data and Encrypted Function

CEF&CED

Scenario

- We'd like to compute function F over data D
  - secret algorithm F or sensitive data D (or both)
- Solution with trusted environment
  - my trusted PC, trusted server, trusted cloud…
- Problem: can be cloud or client really trusted?
  - server hack, DRM, malware...
- Whitebox attacker model
  - controls execution environment (debugging)
  - sees all instructions and data executed

CEF

- Computation with Encrypted Function (CEF)
  - A provides function F in form of P(F)
  - P(F) can be executed on B's machine with B's data D
  - B will not learn function F during its computation (except D_i to $F(D_i)$ mapping)

CED

- Computation with Encrypted Data (CED)
  - B provides encrypted data D as E(D) to A
  - A is able to compute its F as F(E(D)) to produce E(F(D))
    - result of F over D, but encrypted
  - A will not learn data D
  - E(F(D)) is returned back to B and decrypted

CED via homomorphism

1. Convert your function into Boolean circuit with additions (**`xor`**) and multiplications (**`and`**)
2. Compute addition and/or multiplication “securely”
   - an attacker can compute E(D1+D2) = E(D1)+E(D2)
   - but can learn neither D1 nor D2
3. Execute whole circuit over encrypted data

Types of homomorphic schemes

- Partial homomorphic scheme
  - either addition or multiplication is possible, but not both; any number of times
- Somewhat homomorphic scheme
  - Both operations possible, but only limited number of times
- Fully homomorphic scheme
  - both addition and multiplication; unlimited number of times (any computable function)

Partial homomorphic schemes

- Example with RSA (*multiplication*)
  - $E(d_1).E(d_2) = d_1^e \,.\, d_2^e \bmod m = (d_1 d_2)^e \bmod m = E(d_1 d_2)$
- Example Goldwasser-Micali (*addition*)
  - $E(d_1).E(d_2) = x^{d1} r_1^2 \cdot X^{d2} r_2^2 = x^{d1+d2}(r_1 r_2)^2 = E(d_1 \oplus d_2)$
- Limited to polynomial and rational functions
- Limited to only one type of operation (*mult* or *add*)
  - or one type and very limited number of other type
- Slow – based on modular mult or exponentiation
  - every operation equivalent to whole RSA operation

Somewhat Homomorphic Encryption

- Both operations (*mult* and *add*) possible, but only limited number of times
- BGV (Barrat, Gentry and Vaikuntanathan) scheme
- GSW (Gentry-Sahai-Waters) scheme

Fully homomorphic scheme (FHE)

- Holy grail - idea proposed in 1978 (Rivest et al.)
  - both addition and multiplication securely
- But no scheme until 2009 (Gentry)!
- Fully homomorphic encryption
  - based on lattices over integers
  - noisy somewhat homomorphic encryption usable only for few operations
  - combined with repair operation (enable to use it for more operations again)

Fully homomorphic scheme - usages

- Outsourced cloud computing and storage
  - FHE search, Private Database Queries
  - protection of the query content
- Secure voting protocols
  - yes/no vote, resulting decision
- Protection of proprietary info - MRI machines
  - expensive algorithm analyzing MR data, HW protected
  - central processing restricted due to private patient's data
- …

Fully homomorphic scheme - practicality

- Not very practical (yet ☺) (Gentry, 2009)
  - 2.7GB key & 2h computation for every repair operation
  - repair needed every ~10 multiplication
- FHE-AES implementation (Gentry, 2012)
  - standard PC ⇒ 37 minutes/block (but 256GB RAM)
- Gentry-Halevi FHE accelerated in HW (2014)
  - GPU / ASICS, many blocks in parallel => 5 minutes/block
- Replacing AES with other cipher (Simon) (2014)
  - 2 seconds/block
- Very active research area!

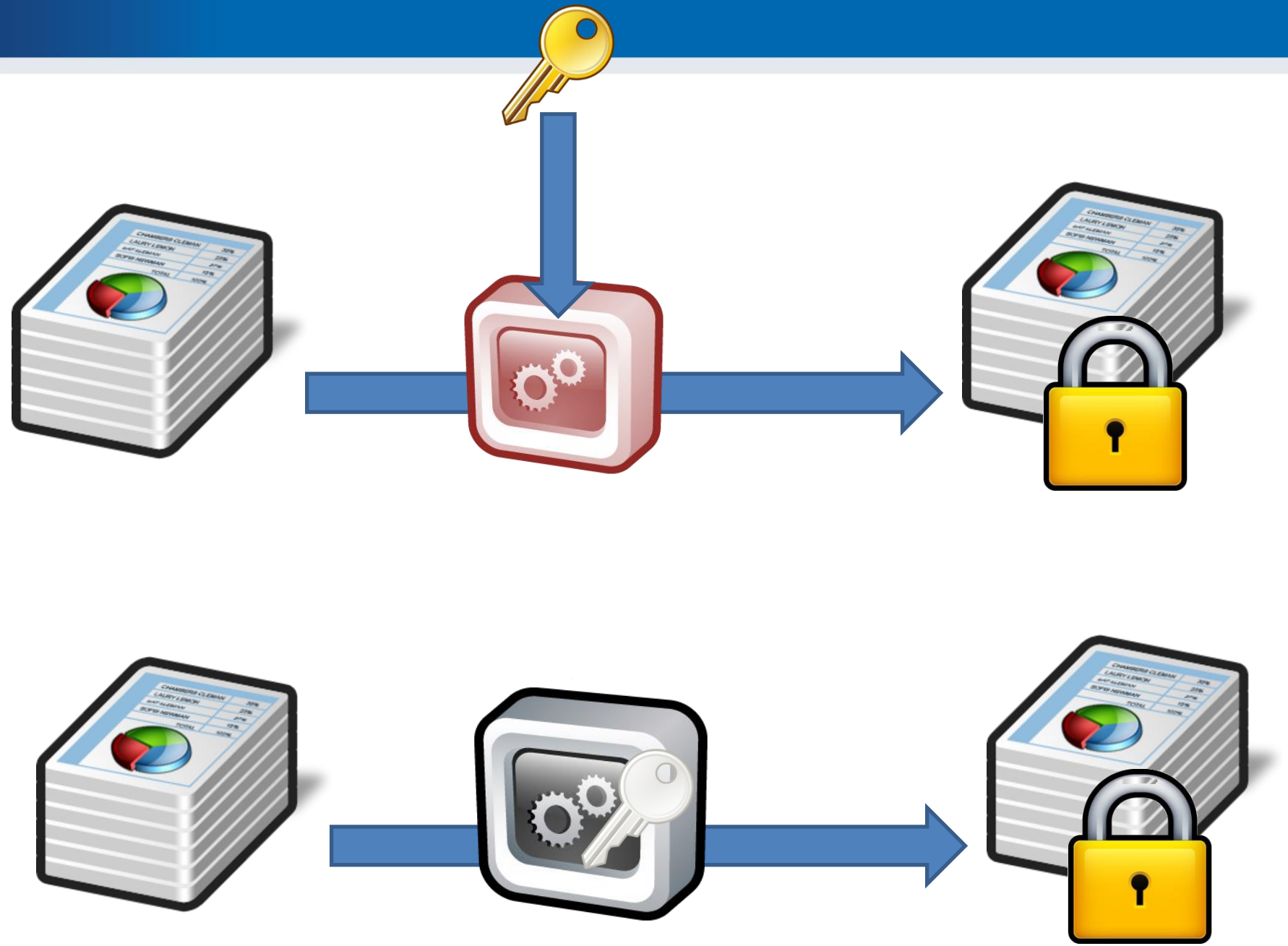
Partial/Fully Homomorphic Encryption libraries

- Homomorphic encryption libraries: HElib, FV-NFLlib, SEAL
- Comparison of features and performance
  - https://arxiv.org/pdf/1812.02428v1.pdf
  - https://link.springer.com/chapter/10.1007/978-3-030-12942-2_32

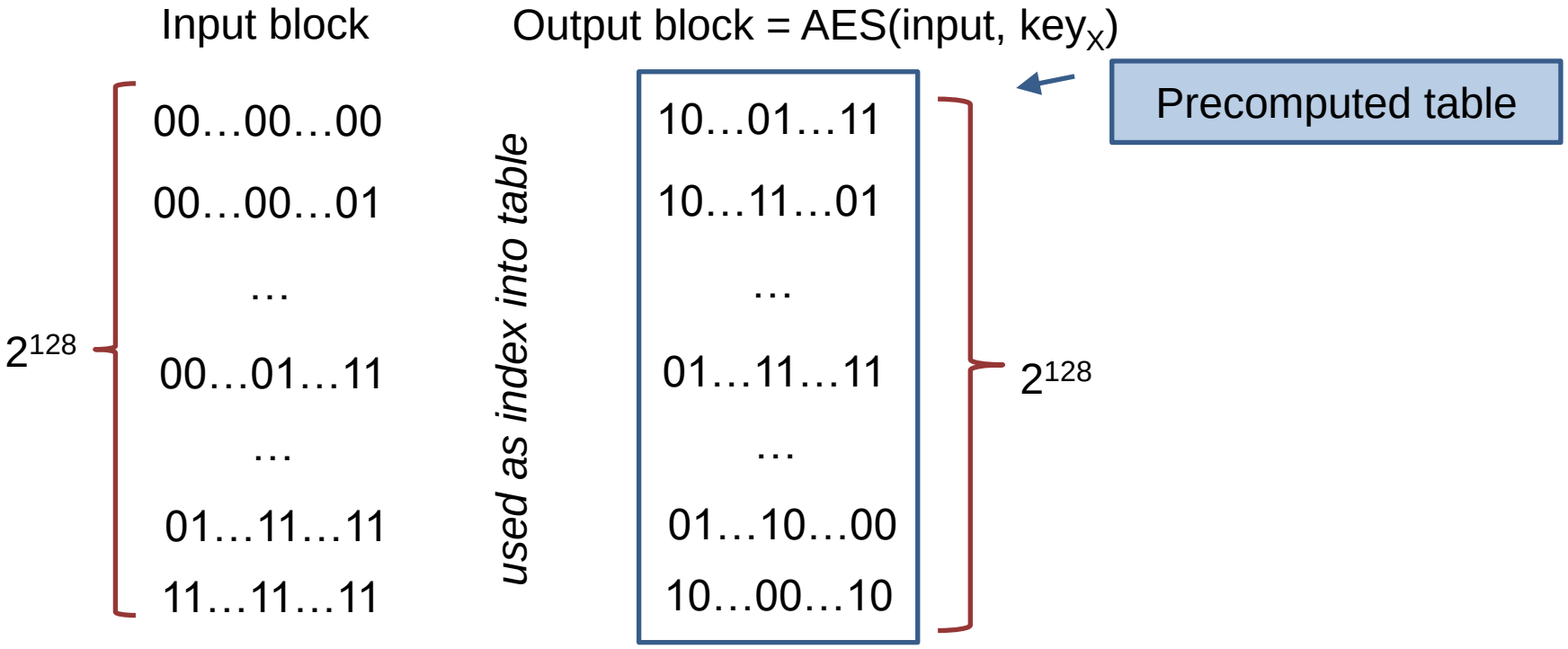
WHITEBOX CRYPTOGRAPHY

White-box attack resistant cryptography

- How to protect symmetric cryptography cipher?
  - protects used cryptographic key (and data)
- Special implementation fully compatible with standard AES/DES… 2002 (Chow et al.)
  - series of lookups into pre-computed tables
- Implementation of AES which takes only data
  - key is already embedded inside
  - hard for an attacker to extract embedded key
  - Distinction between key and implementation of algorithm (AES) is removed



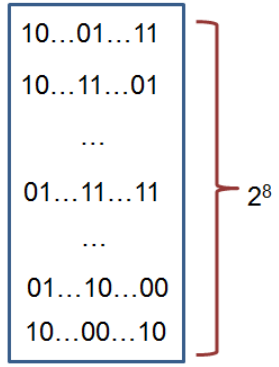
Impractical solution



- Secure, but 2^{128} x 16B memory storage

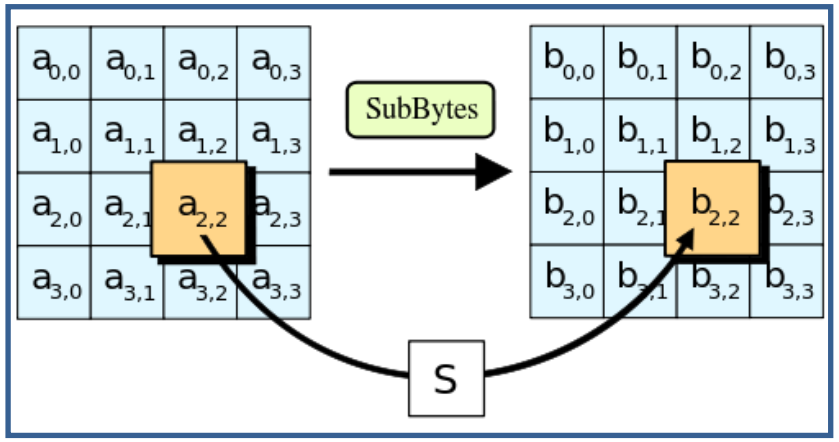
WBACR AES – some techniques

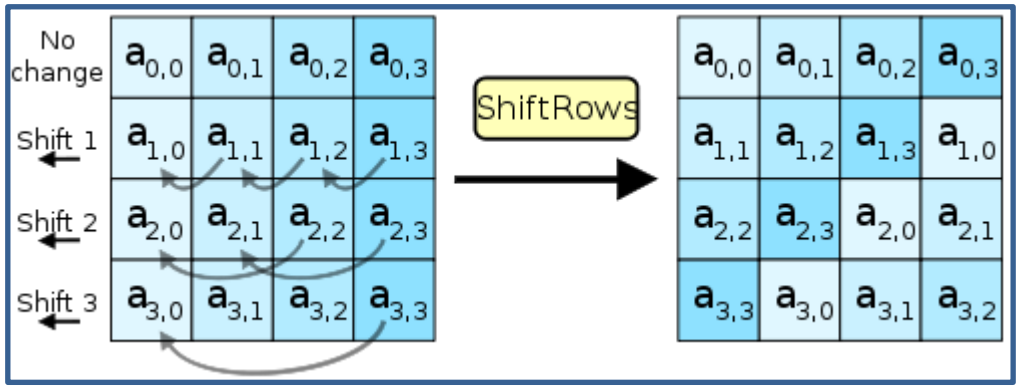
- Pre-compute table for all possible inputs
  - practical for one 16bits or two 8bits arguments table with up to 2^{16} rows (~64KB)
  - **AddRoundKey: data ⊕ key**
    - 8bit argument data, key fixed
- Pack several operations together
  - **AddRoundKey+SubBytes: T[i] = S[i ⊕ key];**
- Protect intermediate values by random bijections
  - removed automatically by next lookup
  - **X = F^{-1}(F(X))**
  - **T[i] = S[F^{-1}(i) ⊕ key];**

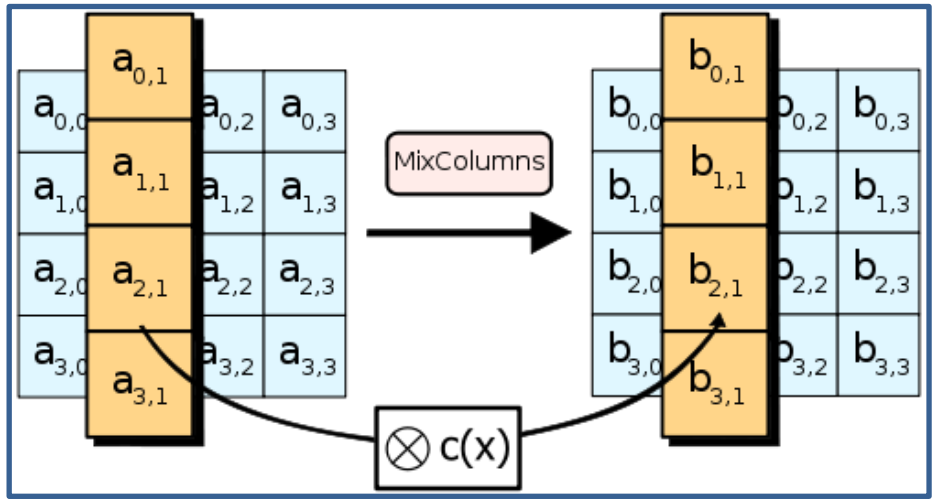


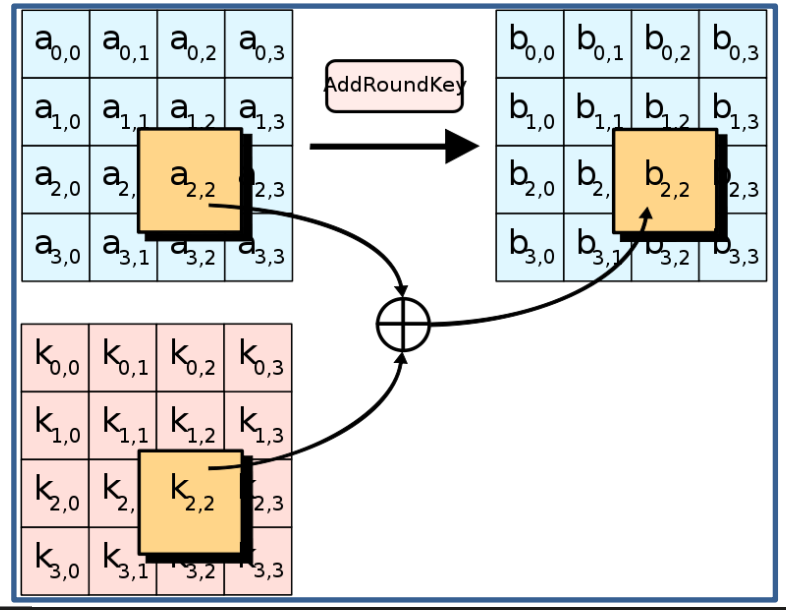
Pictures taken from http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

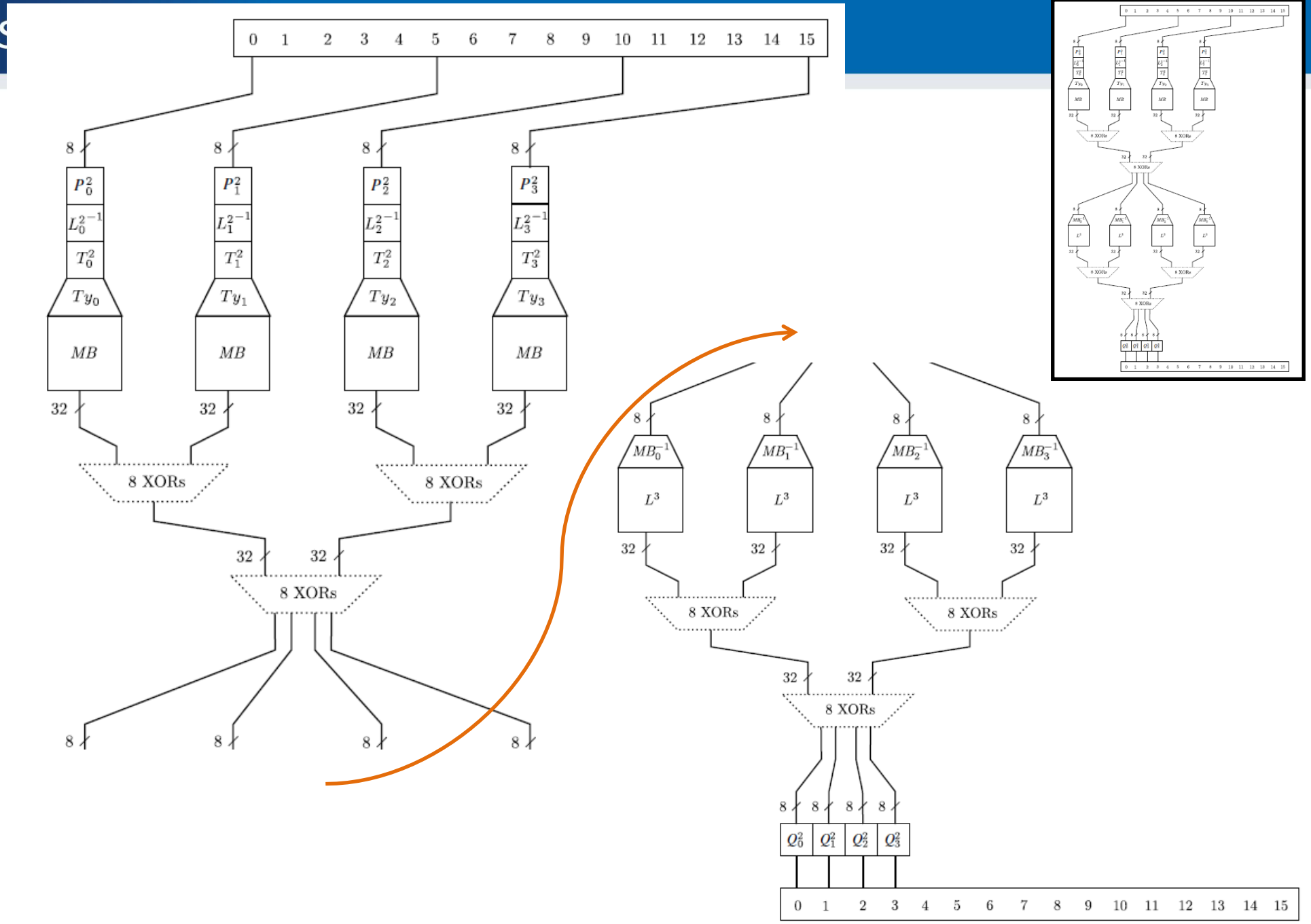
AES – short remainder (used ops)





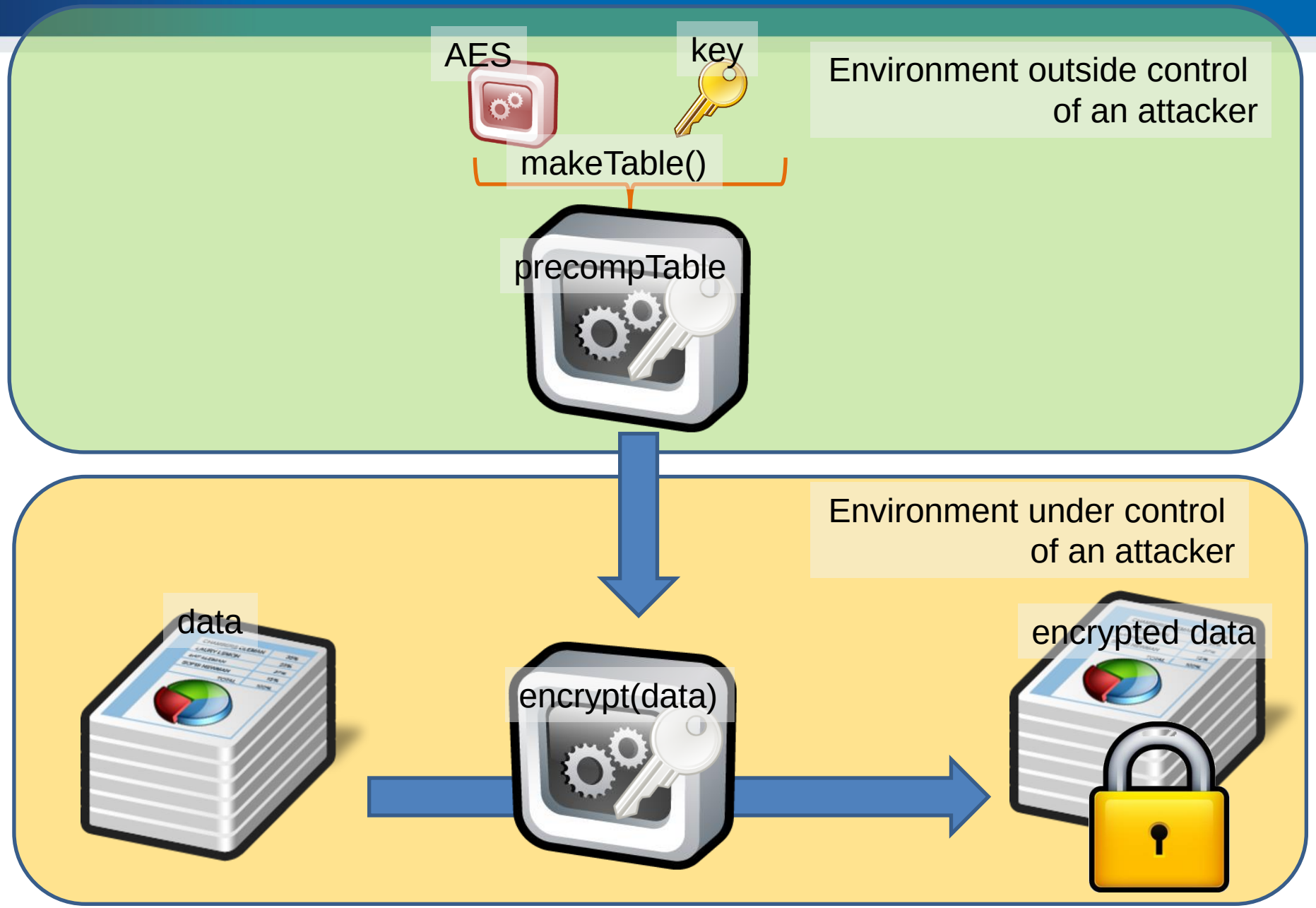





0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
8
P_0^2
L_0^{2-1}
T_0^2
Ty_0
MB
P_1^2
L_1^{2-1}
T_1^2
Ty_1
MB
P_2^2
L_2^{2-1}
T_2^2
Ty_2
MB
P_3^2
L_3^{2-1}
T_3^2
Ty_3
MB
32
8 XORs
8 XORs
8 XORs
MB_0^{-1}
MB_1^{-1}
MB_2^{-1}
MB_3^{-1}
L^3
Q_0^2 Q_1^2 Q_2^2 Q_3^2

Whitebox cryptography lifecycle

- [Secure environment]
  1. Generate required key (random, database...)
  2. Generate WAES tables (in secure environment)
- [Potential insecure environment]
  3. Compile WAES tables into target application
- [Insecure environment (User PC)]
  4. Run application and use WAES as usual (with fixed key)


AES
key
Environment outside control of an attacker
makeTable()
precompTable
Environment under control of an attacker
data
encrypt(data)
encrypted data

Resulting implementation

- More difficult to detect that crypto was used
  - no fixed constants in the code
  - precomputed tables change for every new AES instance
    - even two tables for same key are different
  - (but can still be detected)
- Resistant even when precomputed tables are found
  - when debugged, only table lookups are seen
  - key value is never manipulated in plaintext
  - transformation techniques should provide protection to key embedded inside tables

Demo – WAES

- WAES tables generator
  - configuration options
  - *.h files with pre-computed tables
- WAES cipher implementation
  - compile-in tables
  - tables as memory blob

```
# Encryption and decryption key values. Used during "/g:" option .
[KEY_VALUE]
# Key value which will be used for encryption part of WBACR AES tables.
# CAN be different from decryptKey
encryptKey=8a b1 21 d3 13 d1 5e 31 29 84 4c 66 50 14 6e 95
# Key value which will be used for decryption part of WBACR AES tables.
decryptKey=00 01 02 03 05 06 07 08 0A 0B 0C 0D 0F 10 11 12
# Additional entropy used for WBACR DES tables pre-computation
entropy=4d 28 a7 cd f9 c6 64 bc 94 c4 0e 77 79 7a 41 dc d8 19 9a 4b 0e 1c 74 88 a5
```

```
#ifndef AESINVFIRSTTABLE_AUTH_H
#define AESINVFIRSTTABLE_AUTH_H

BYTE            invFirstRoundTable_auth[4][4][256] = {
{
{0x23, 0xee, 0x4c, 0x94, 0x4e, 0x32, 0x95, 0x3d, 0xa6, 0x
0xff, 0x34, 0x8e, 0x39, 0xcb, 0x82, 0x43, 0x87, 0x9b, 0xe
0xf, 0xc1, 0xaf, 0x1e, 0x6b, 0x8f, 0xbd, 0x2, 0xca, 0x8a,
0xc7, 0xb1, 0x12, 0xa8, 0x5f, 0x33, 0x10, 0x31, 0x88, 0xe
0xd6, 0xe1, 0x69, 0x4, 0x7d, 0x7e, 0x14, 0x26, 0xba, 0xc,
0x35, 0xe2, 0xf9, 0x74, 0x6e, 0x22, 0x37, 0x85, 0xe7, 0xc
0x24, 0x76, 0x5b, 0xa1, 0x25, 0x66, 0xa2, 0xb2, 0x28, 0xc
0x79, 0x9a, 0xdb, 0x3e, 0xf4, 0x4b, 0xc0, 0x20, 0xc6, 0x
0xf5, 0xc8, 0xeb, 0x3b, 0x61, 0x4d, 0xbb, 0xb0, 0xae, 0x
0xb9, 0x5e, 0x15, 0x48, 0x84, 0x50, 0x46, 0xda, 0xfd, 0x
```

WAES performance

- Intel Core i5 M560@2.67GHz

Test	Result	Additional info.	OpenSSL result
generate WB AES	8.48 s avg.	100 samples	
throughput, 1 MB random	867.8 KB/s	1.18 s	57283 KB/s
throughput, 10 MB random	1022.977 KB/s	10.01 s	54179 KB/s
throughput, 100 MB random	1028.319 KB/s	99.58 s	74744 KB/s
throughput, 1024 MB random	1124.792 KB/s	932.24 s	63723 KB/s
throughput, 1 MB null	975 KB/s	1.05 s	93091 KB/s
throughput, 10 MB null	969.970 KB/s	10.56 s	68821 KB/s
throughput, 100 MB null	1058.507 KB/s	96.74 s	56356 KB/s
throughput, 1024 MB null	1050.593 KB/s	998.08 s	57283 KB/s

L. Bacinska, https://is.muni.cz//th/373854/fi_m/

WBACR Ciphers - pros

- Practically usable (size/speed)
  - implementation size ~800KB (WBACR AES tables)
  - speed ~MBs/sec (WBACRAES ~6.5MB/s vs. 220MB/s)
- Hard to extract embedded key
  - Complexity semi-formally guaranteed (if scheme is secure)
  - AES shown unsuitable (all WBARC AESes are broken)
- One can simulate asymmetric cryptography!
  - implementation contains only encryption part of cipher
  - until attacker extracts key, decryption is not possible

WBACR Ciphers - cons

- Implementation can be used as oracle (black box)
  - attacker can supply inputs and obtain outputs
  - even if she cannot extract the key
  - (can be partially solved by I/O encodings)
- Problem of secure input/output
  - protected is only cipher (e.g., AES), not code around
- Key is fixed and cannot be easily changed
- Successful cryptanalysis for several schemes ☹
  - several former schemes broken
  - new techniques being proposed

Space-Hard Ciphers

- Space-hard notion of WBACR ciphers
  - How much can be fnc compressed after key extraction?
    - WBACR AES=>16B key=>extreme compression (bad)
  - Amount of code to extract to maintain functionality
- SPACE suite of space-hard ciphers
  - Combination of l-line target heavy Feistel network and precomputed lookup tables (e.g., by AES)
  - Variable code size to exec time tradeoffs

Can whitebox transform replace secure hardware (e.g., smart card)?

- Only to limited extent
- Limitation of arguments size
- Operation atomicity
  - one cannot execute only half of card's operations
- No secure memory storage
  - no secure update of state (counter)
- Both can be used as black-box
  - smart card can use PIN to limit usage
- But still some reasonable usages remain

List of proposals and attacks

- (2002) First WB AES implementation by Chow et. al. [Chow02]
  - IO bijections, linear mixing bijections, external coding
  - broken by BGE cryptanalysis [Bill04]
    - algebraic attack, recovering symmetric key by modelling round function by system of algebraic equations
- (2006) White Box Cryptography: A New Attempt [Bri06]
  - attempt to randomize whitebox primitives, perturbation & random equations added, S-boxes are enc. keys. 4 AES ciphers, major voting for result
  - broken by Mulder et. al. [Mul10]
    - removes perturbations and random equations, attacking on final round removing perturbations, structural decomposition. 2^{17} steps
- (2009) A Secure Implementation of White-box AES [Xia09]
  - broken by Mulder et. al. [Mul12]
    - linear equivalence algorithm used (backward AES-128 compatibility => linear protection has to be inverted in next round), 2^{32} steps
- (2011) Protecting white-box AES with dual ciphers [Kar11]
  - broken by our work [Kli13]
    - protection shown to be ineffective
- Fault induction especially devastating

BGE attack in progress

```
   recoverQj; q = 0x88; gamma=0x01;
   recoverQj self-test; r=5; col=3; (y0, y3); P[0].deltaInv=0x03; alfa_{3,0}=0x03
   recoverQj self-test; r=5; col=3; (y0, y3); P[1].deltaInv=0x01; alfa_{3,1}=0x01
   recoverQj self-test; r=5; col=3; (y0, y3); P[2].deltaInv=0x01; alfa_{3,2}=0x01
   recoverQj self-test; r=5; col=3; (y0, y3); P[3].deltaInv=0x02; alfa_{3,3}=0x02
   recoverQj; q = 0x3c; gamma=0x01;

Going to reconstruct encryption key from extracted round keys...
* Round keys extracted from the process, r=3
 0x3d  0x47  0x1e  0x6d  0x80  0x16  0x23  0x7a  0x47  0xfe  0x7e  0x88  0x7d  0x3e  0x44  0x3b

* Round keys extracted from the process, r=4
 0xef  0xa8  0xb6  0xdb  0x44  0x52  0x71  0x0b  0xa5  0x5b  0x25  0xad  0x41  0x7f  0x3b  0x00

* Round keys extracted from the process, r=5
 0xd4  0x7c  0xca  0x11  0xd1  0x83  0xf2  0xf9  0xc6  0x9d  0xb8  0x15  0xf8  0x87  0xbc  0xbc

Recovering cipher key from round keys...
We have correct Rcon! rconIdx=3
RC=2; previousKey:
 0xf2  0x7a  0x59  0x73
 0xc2  0x96  0x35  0x59
 0x95  0xb9  0x80  0xf6
 0xf2  0x43  0x7a  0x7f

RC=1; previousKey:
 0xa0  0x88  0x23  0x2a
 0xfa  0x54  0xa3  0x6c
 0xfe  0x2c  0x39  0x76
 0x17  0xb1  0x39  0x05

RC=0; previousKey:
 0x2b  0x28  0xab  0x09
 0x7e  0xae  0xf7  0xcf
 0x15  0xd2  0x15  0x4f
 0x16  0xa6  0x88  0x3c

Final result:
 0x2b  0x7e  0x15  0x16  0x28  0xae  0xd2  0xa6  0xab  0xf7  0x15  0x88  0x09  0xcf  0x4f  0x3c

Benchmark finished! Total time = 3a s; on average = 58 s; clocktime=57.66 s;
```

More resources

- Overviews, links
  - http://whiteboxcrypto.com/research.php
  - https://minotaur.fi.muni.cz:8443/~xsvenda/docuwiki/doku.php?id=public:mobilecrypto
- Crackme challenges
  - http://www.phrack.org/issues.html?issue=68&id=8
- Whitebox crypto in DRM
  - http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf

Whitebox transform IS used in the wild

- Proprietary DRM systems
  - details are usually not published
  - AES-based functions, keyed hash functions, RSA, ECC...
  - interconnection with surrounding code
- Chow at al. (2002) proposal made at Cloakware
  - firmware protection solution
- Apple's FairPlay & Brahms attack
  - http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf
- ...